Über den Entwurf von Benutzungsschnittstellen technischer Anwendungen mit visuellen Spezifikationsmethoden und Werkzeugen

Endbenutzeraspekte
Wiederverwendung durch Entwurfsmuster und Komponenten

Vom Fachbereich Informatik
der TechnischenUniversität Darmstadt genehmigte

Dissertation

zum Erlangen des akademischen Grades des
Doktor-Ingenieurs (Dr.-Ing.)

vorgelegt von
Dipl.-Inform. Elke Siemon
geboren in Wolfenbüttel

Referenten und Referentin der Arbeit:
Prof. Dr.-Ing. H.-J. Hoffmann
Prof. Dr. rer. nat. G. Szwillus
Prof. Dr.-Ing. M. Mezini

Tag des Einreichens: 12. Januar 2001
Tag der mündlichen Prüfung: 29. Januar 2001

D17

Vorwort

Kognitive Modelle (d. h. die Vorstellung von Menschen) über Computersysteme sind in vielen Bereichen gut erforscht. Beispielsweise ist die Vorstellung über die *Benutzung* von Computersystemen Gegenstand der Software-Ergonomie und der Mensch-Maschine-Forschung (engl. *human computer interaction*, HCI). In der Software-Entwicklung gibt es viele Untersuchungen über die Vorstellungen der Programmierer/innen beim *Entwickeln von Software*. Auch die Anfangsphasen der Entwicklung der Mensch-Maschine-Schnittstellen (*Benutzungsschnittstellen*) werden durch Modelle der Forschung unterstützt, z. B. gibt es geprüfte Vorgehensmethoden zur Ermittlung der Aufgaben der zukünftigen Benutzer/in einer Mensch-Maschine-Schnittstelle.

Dagegen ist der Bereich der Vorstellungen bei der *Software-Entwicklung von Benutzungsschnittstellen* allgemein noch nicht durch konzeptuelle Modelle fundiert. Die Vorgehensweisen, die durch *Werkzeuge zur Entwicklung von Benutzungsschnittstellen* unterstützt werden, basieren auf der Vorstellung der jeweiligen Werkzeug-Gestalter/in. Diese Vorstellung hängt von verschiedenen Faktoren ab, z. B.:

- Von der *Fachdisziplin*, der die Werkzeug-Gestalter/in angehört, z. B. Software Engineering, Visuelle Programmiersprachen oder auch der Anwendungsgebiete, in denen die Benutzungsschnittstellen eingesetzt werden sollen.
- Von der Kenntnis der Werkzeug-Gestalter/in über die Entwicklung von Benutzungsschnittstellen. Das Wissen über Verhaltensmechanismen, Struktur und Architektur ist auch für die *Gestaltung und damit die Benutzung des Werkzeugs* von Bedeutung. Oft kennen Werkzeug-Gestalter/innen nur ein Benutzungsschnittstellen-Prinzip (beispielsweise das Callback-Prinzip, das später erklärt wird), und das oft auch nur in einer Sprache und für eine konkrete Klassenbibliothek. Deshalb gibt es häufig Verständnisschwierigkeiten zwischen Entwickler/innen, die verschiedene Werkzeuge benutzen.

In dieser Arbeit werden daher die *Vorstellungen bei der Entwicklung von Benutzungsschnittstellen auf einer höheren Ebene*, d. h. *implementierungsübergreifend, fachgebietsübergreifend* und *übergreifend über konkrete Mechanismen*, anhand von *drei* (in dieser Arbeit durchgeführten) *Untersuchungen* erforscht. Die Erkenntnisse dieser Forschung werden in einem neuen Modell zusammengefaßt, das die übergreifenden Vorstellungen durch eine einheitliche Terminologie ausdrückt: dem *Aspektmodell*. Das Aspektmodell ist für Software-Ingenieur/innen, Designer/innen von Benutzungsschnittstellen und Anwendungs-Expert/innen verständlich. Weiterhin werden Abstraktionsebenen der Programmierung unterschieden, die Basiskonzepte, Konzepte der Software-Wiederverwendung und aufgabenorientierte Konzepte berücksichtigen. Diese Ebenen finden sich in der textuellen und der visuellen Programmierung wieder. Es zeigt sich, daß das Aspektmodell in jeder Ebene gültig ist und die klare gedankliche Abgrenzung zwischen den Ebenen sowie das gedankliche Auf- und Absteigen zwischen den Ebenen bei der Programmierung unterstützt, das für allgemeingültige Programmiertechniken nicht vermieden werden kann.

Das Aspektmodell ist ebenfalls die Grundlage für das von mir in dieser Arbeit entworfene und implementierte *Werkzeug COMBO*. Wegen dieser Basis ist COMBO erwartungsgemäß für Entwickler/innen verschiedener Disziplinen verständlich und erlaubt eine leichtere Verständigung zwischen den Entwickler/innen der unterschiedlichen Fachrichtungen.

Die *zur Umsetzung des Aspektmodells notwendigen technischen Grundlagen* wurden ebenfalls in dieser Arbeit erforscht:

- Es wird herausgearbeitet, welchen *Prinzipien* ein Werkzeug genügen muß, das das Aspektmodell umsetzt.
- Es werden *visuelle Spezifikationsmethoden* entworfen und implementiert, mit denen die Aspekte des Aspektmodells beschrieben werden können. Aus den Spezifikationen wird *Programmtext generiert*, der, zusammengesetzt, eine lauffähige Benutzungsschnittstelle generiert. Weitere, bereits *existierende Spezifikationsmethoden werden in das Werkzeug COMBO integriert.*
- Es wird untersucht, wie Konzepte des Software Engineering zur *Wiederverwendung*, und zwar *Anwendungsrahmen, Entwurfsmuster und Komponenten*, unterstützt werden können. Aufbauend auf dem Aspektmodell werden (im Rahmen dieser Arbeit gefundene) *neue Entwurfsmuster für Benutzungsschnittstellen* und visuelle Spezifikationsmethoden für Anwendungsrahmen diskutiert und für Entwurfsmuster und Komponenten vorgestellt und realisiert.

Danksagung

Die vorliegende Arbeit begann in Assoziation mit dem Graduiertenkolleg „Intelligente Systeme für die Informations- und Automatisierungstechnik", gefördert durch die Promotionsförderung „Frauen in den Ingenieursdisziplinen". Sie wurde danach während der Zeit als wissenschaftliche Mitarbeiterin am Fachgebiet Programmiersprachen und Übersetzer im Fachbereich Informatik der Technischen Universität Darmstadt fertiggestellt.

Ich wurde von Herrn Prof. Dr. H.-J. Hoffmann betreut, und ich möchte ihm für die Unterstützung meiner Forschungstätigkeit, die zum Gelingen dieser Arbeit beitrug, danken.

Herrn Prof. Dr. G. Szwillus und Frau Prof. M. Mezini danke ich für regelmäßige Ermutigungen und die Übernahme des Koreferats.

Mein Dank gilt allen Mitarbeiterinnen und Mitarbeitern der Fachgebiete „Programmiersprachen und Übersetzer" und „Simulation" für die kollegiale Hilfe bei den kleinen und großen Problemen der täglichen Arbeit und kritische fachliche Diskussionen. Die Unterstützung von Prof. Dr. von Stryk am Schluß der Arbeit hat sehr gut getan. Besonderer Dank auch an Yongmei Wu für die gute Zusammenarbeit sowie an Ludger Martin als „Extreme Programming[1]"-Partner.

Ich möchte mich auch bei allen Studentinnen und Studenten bedanken, die im Rahmen von Studien- und Diplomarbeiten unter meiner (Mit-)Betreuung ebenfalls zum Gelingen dieser Arbeit beigetragen haben: Christian Begger, Martin Hartmann, Wolfgang Hess, Dirk Jacob, Ralph Juhnke, Ursula Jütten, Marlis Koch, Ludger Martin, Jens Püttmann, Michael Renker, Thorsten Rottschäfer, Torsten Scheidler und Markus Zahnjel.

Meiner Mutter danke ich für die mehrfache Durchsicht und Korrektur der Arbeit ganz besonders.

Diese Arbeit wäre nicht beendet worden ohne die Unterstützung und Motivation vieler Freunde, die an mich geglaubt haben, und die mir viel Zeit für wissenschaftliche und persönliche

[1] [Bec00]

Diskussionen geschenkt haben: Meine Eltern, Andrea Gorlt, Eva Stephan, Gisela Neumann, Jan Weerts, Günter Brast, Christine Schanz und Volker Jung, Verena Willenbrinck und Ute Blotenberg, Jochen Koubek und (fast alle) von Git on boa'd.

Zur Schreibweise

In der Informatik gibt es für viele Konzepte keine einheitliche Terminologie, insbesondere in neueren Techniken. Oft werden anderen Forschungsgebieten Begriffe entlehnt, wenn die Konzepte dort ähnlich sind. Daneben gibt es oft das Problem, daß englische Fachtermini entweder gar nicht oder uneinheitlich übersetzt werden. Als Beispiel sei das Wort Benutzungsschnittstelle genannt, das in der Literatur oft durch (graphische) Benutzungsoberfläche, user interface, interface, Benutzerschnittstelle, Benutzer-Interface, Benutzeroberfläche, Mensch-Maschine-Schnittstelle, human computer interface usf. ersetzt wird.

In dieser Arbeit werden, wenn möglich, deutsche Begriffe verwendet. Die wichtigsten in dieser Arbeit verwendeten Begriffe werden bei der ersten Verwendung kurz erklärt, bzw. im Glossar 8 aufgeführt.

Die Technische Universität Darmstadt (TUD) hat sich zum Ziel gesetzt, den Frauenanteil zu erhöhen, insbesondere in den Fachbereichen, in denen Frauen unterrepräsentiert sind (z. B. lag der Frauenanteil in der Informatik in den letzten Jahren bei ca. 10-20%). Daher werden besondere Anstrengungen unternommen, Frauen als Mitglieder des Fachbereichs (Professorinnen, Mitarbeiterinnen, Studentinnen) zu gewinnen. Weitere Projekte, die von den Frauen am Fachbereich und der TUD initiiert und durchgeführt wurden, haben das gleiche Ziel.

In diesem Sinne ist es mir ein besonderes Anliegen, in meinen Texten darauf hinzuweisen, daß gerade die Informatik auch Frauensache ist. *Eine* Möglichkeit dazu ist, für Personenbezeichnungen in meinen Texten durchgängig eine gemischte Schreibweise der Form Programmierer/innen und Entwickler/innen zu verwenden. Damit soll jedesmal, wenn eine solche Personenbezeichnung verwendet wird, darauf hingewiesen werden, daß jede der Tätigkeiten (z. B. Programmieren, Entwickeln, etc.) hoffentlich möglichst bald von so vielen Frauen wie Männern durchgeführt wird.

Ich hoffe, die Leserinnen und Leser verzeihen den verringerten Lesekomfort und haben trotzdem Spaß beim Lesen.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Motivation

Heute ist der Computer an jedem Arbeitsplatz präsent und deshalb wird die Arbeitsumgebung des Menschen wesentlich durch die Gestaltung der Schnittstelle zwischen Mensch und Computer (im folgenden *Benutzungsschnittstelle* genannt) geprägt.

Seit ihrer Erfindung in den 60er und 70er Jahren sind *graphische Benutzungsschnittstellen* zunehmend leichter bedienbar geworden.
Auch die Werkzeuge zu ihrer Erstellung wurden weiter entwickelt, von

- *Expert/innen für Mensch-Maschine-Interaktion* (engl. *human computer interaction*, *HCI*),
- von *Software-Ingenieur/innen* und von
- *Anwendungs-Expert/innen.*

Leider haben Entwickler/innen dieser Werkzeuge selten zusammengearbeitet, so daß es in jedem Bereich unterschiedliche Werkzeuge gibt. Laut Myers [Mye00c] sind Hunderte von Benutzungsschnittstellen-Entwicklungswerkzeugen kommerziell verfügbar, und noch viel mehr sind als Prototypen in Forschungseinrichtungen entstanden. In dieser Arbeit werden die Ansätze der genannten Gebiete untersucht und zusammengeführt.

Benutzungsschnittstellen von Büro-, Datenbank- und Web-Anwendungen bestehen inzwischen aus ausprogrammierten Standardelementen, die bei der Konstruktion neuer Benutzungsschnittstellen wiederverwendet werden können.

Und doch gibt es Bereiche, in denen Benutzungsschnittstellen auch heute noch von „ganz unten", d. h. von Hand, programmiert werden müssen. Das ist nicht verwunderlich, wenn es sich um noch nicht etablierte Anwendungen handelt (z. B. Anwendungen im mobilen Bereich, der simulierten Realität, engl. *virtual reality* oder der erweiterten Realität, engl. *augmented reality*).

Es ist erstaunlich, daß auch die etablierten zweidimensionalen objektorientierten *Benutzungsschnittstellen technischer Anwendungen* dazugehören, z. B. Bediensysteme von Fabrikanlagen oder Prozeßautomatisierungs-Systeme (im nächsten Abschnitt werden Beispiele vorgestellt). Wünschenswert sind Werkzeuge oder Methoden, mit denen die Benutzungsschnittstelle spezifiziert, d. h. auf einer höheren Abstraktionsstufe definiert werden kann.

Für technische Anwendungen gibt es keine „optimalen" Benutzungsschnittstellen-Entwicklungswerkzeuge in dem Sinne, daß sie

- von der Expert/in eines Anwendungsgebiets *ohne Programmierkenntnisse* benutzbar sind,
- über das Verwenden vordefinierter Benutzungsschnittstellen-Elemente hinausgehende *flexible Spezifikationen* von Benutzungsschnittstellen anbieten sowie
- neue Erkenntnisse der Informatik zur *Wiederverwendung* einbeziehen, z. B. durch Verwenden von Anwendungsrahmen, Entwurfsmustern und Komponenten.

Daher liegt der Schwerpunkt dieser Dissertation auf Methoden und Werkzeugen zur Entwicklung von Benutzungsschnittstellen technischer Systeme: Es wird untersucht, welche *Rolle Endbenutzer/innen im Entwicklungsprozeß* spielen und welche *Anforderungen* sie *an ein Werkzeug* stellen. Daraus wird ein neues Modell abgeleitet, das *Aspektmodell*, das unabhängig vom Anwendungsgebiet zur Beschreibung von Benutzungsschnittstellen dient. In dieser Arbeit wurde das auf dem Aspektmodell beruhende Werkzeug COMBO von mir entworfen und als Prototyp realisiert.

COMBO bietet Spezifikationsmethoden sowohl für die Endbenutzer/in (vollständig visuell und aufgabenorientiert) als auch für die Entwickler/in mit Programmierkenntnissen (visuell auf der Ebene von Strukturen und Mechanismen der Klassenbibliotheken oder auch textuell) an.

1.1 Problematik, Ziel der Arbeit und Lösungsansatz, illustriert durch Beispiele

Die vorliegende Problematik soll an zwei Beispielen verdeutlicht werden.

Am ersten Beispiel wird gezeigt, daß bereits der Entwurf einfacher Bildschirmelemente schwierig ist und wie eine intuitive Spezifikationsmethode aussehen könnte. Eine *Spezifikationsmethode* beschreibt Notation und Vorgehen zur Spezifikation eines Programms oder Systems.

Das zweite Beispiel zeigt komplexere Benutzungsschnittstellen, die aus mehreren einfachen Bildschirmelementen aufgebaut sind.

Beispiel 1: Ein Tank mit Zu- und Ablauf

Benutzungsschnittstellen für die Bedienung technischer Anlagen stellen oft Teile der Anlage visuell dar.

Für eine Kläranlage beispielsweise könnten der Füllstand eines Mischbeckens und die Sauerstoffkonzentration durch ein graphisches Tank-Element dargestellt werden, wie in Abbildung 1.1 gezeigt.

Der Füllstand wird durch die Füllhöhe des graphischen Tank-Elements repräsentiert und die momentane Sauerstoffkonzentration durch die Farbe des Inhalts (z. B. rot = Sauerstoffkonzentration zu niedrig, grün = ok, blau = zuviel Sauerstoff).

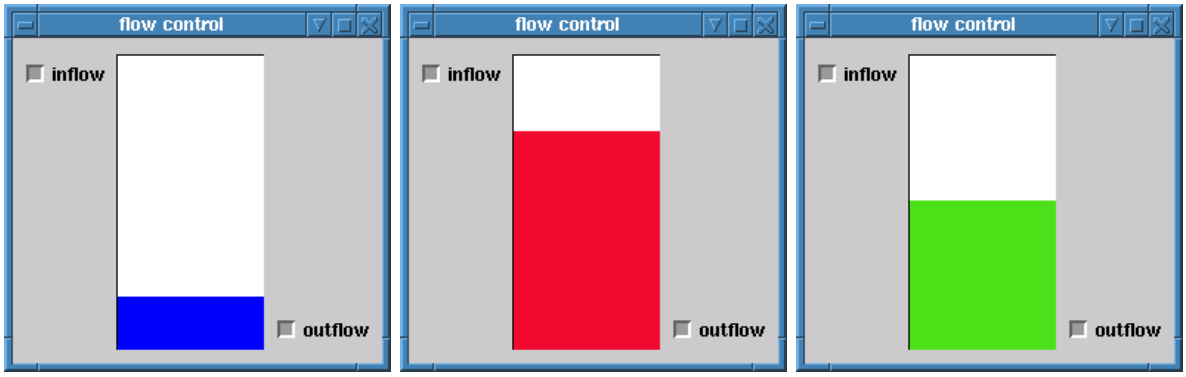


Abbildung 1.1: Bildschirmelement zur Darstellung des Inhalts eines Tanks in den unterschiedlichen Zuständen (links: blau = zuviel Sauerstoff, Mitte: rot= zuwenig Sauerstoff, rechts: grün = ok)

Problematik

Viele Werkzeuge basieren darauf, daß vorgefertigte Elemente verwendet werden können, die sehr anschaulich und intuitiv angeordnet werden können.

Beliebige graphische Elemente, wie das vorgestellte Tank-Element, sind nicht als vorgefertigte Bausteine vorhanden, sondern müssen „von Hand" programmiert werden.

Das Programmieren „von Hand" in einer Programmiersprache wie C, C++ oder Java erfordert das Erlernen sowohl der Programmiersprache als auch der entsprechenden Klassenbibliotheken und entsprechend breites Informatik-Wissen.

Ziel

Die Lücke zwischen der Benutzung vorgefertigter Elemente und der Programmierung „von Hand" soll mit der vorliegenden Arbeit geschlossen werden: Dazu müssen eine *Vorgehensweise* und ein *Werkzeug* entwickelt werden, die *anschauliche und intuitive Programmierung beliebiger graphischer Elemente* ermöglichen.

Um dieses Ziel zu erreichen, müssen folgende *Fragen* beantwortet werden:

- Was ist *intuitive,* d. h. „auf unmittelbarer Anschauung beruhende" [WB99], Spezifikation von Benutzungsschnittstellen?
- Was ist das Besondere beim Entwurf von Benutzungsschnittstellen in dem *Anwendungsgebiet „Benutzungsschnittstellen technischer Systeme"*?
- Welche *Aufgaben* treten *bei der Programmierung* einer Benutzungsschnittstelle auf, die durch eine intuitive Spezifikation vereinfacht werden könnten?
  - Wie kann das *Aussehen* der einzelnen Benutzungsschnittstellen-Elemente intuitiv spezifiziert werden?
  - Wie kann das *Verhalten* der einzelnen Benutzungsschnittstellen-Elemente intuitiv spezifiziert werden?
  - Wie wird die *Verbindung* zur Anwendung intuitiv hergestellt?

Die Fragen lassen sich nicht alleine mit den Mitteln der Informatik beantworten, sie streifen auch Gebiete, die sich mit menschlichen Vorstellungen und Fähigkeiten beschäftigen, sowie das Anwendungsgebiet. Daher wird folgender Lösungsansatz gewählt:

Interdisziplinäres Vorgehen
Zunächst werden die Erkenntnisse der Gebiete (*Kognitions-)Psychologie*, *technische Anwendungen* und *Informatik* untersucht. Mit Hilfe von Untersuchungmethoden und Vorgehensmodellen werden Antworten auf die genannten Fragen abgeleitet. Aus den Antworten wird ein Modell zur Entwicklung von Benutzungsschnittstellen entwickelt, das als Grundlage für ein prototypisches Werkzeug dient.

Als eine mögliche Antwort auf die Frage nach intuitiver Spezifikation wird - in Vorwegnahme späterer Ergebnisse - in Abbildung 1.3 ein Beispiel für eine „intuitive Spezifikation" mit Papier und Bleistift gezeigt, die mehrfach im Rahmen studentischer Arbeiten verwendet wurde, um die Tank-Benutzungsschnittstelle zu erklären.

Wie das Beispiel zeigt, scheinen Menschen visuelle Darstellungen von Systemen gerne zu benutzen, wenn sie über Benutzungsschnittstellen reden[1]. Zu beobachten ist auch, daß in der Regel das System in mehreren Ansichten dargestellt wird, die dann durch Pfeile und Erklärungen ergänzt werden. Die Pfeile und Erklärungen beschreiben damit die Semantik der Benutzungsschnittstelle.

Es gibt noch kein Werkzeug, das diese Spezifikation vom Papier in ein Programm umsetzt! Das liegt daran, daß es zwar Werkzeuge gibt, mit denen man Skizzen von Papier einlesen kann und auch Werkzeuge, die Gesten erkennen können, wenn diese Skizze mit einem Stift als Eingabegerät gezeichnet wird [LM00, DHT00]. Es gibt aber keine Werkzeuge, die diese Pfeile in Programme umsetzen, d. h. die Semantik der Zeichnung verstehen. Das in dieser Arbeit prototypisch implementierte Werkzeug COMBO versteht die Semantik und kann die Pfeile interpretieren, simuliert jedoch „Papier und Stift" durch Maus- und Tastatur-Interaktionen.

Entwuf einer Benutzungschnittstelle für einen Tank

Zuerst wurden die Elemente der Anlage, der Sensor-Datenverarbeitung und der Benutzungsschnittstelle dargestellt:

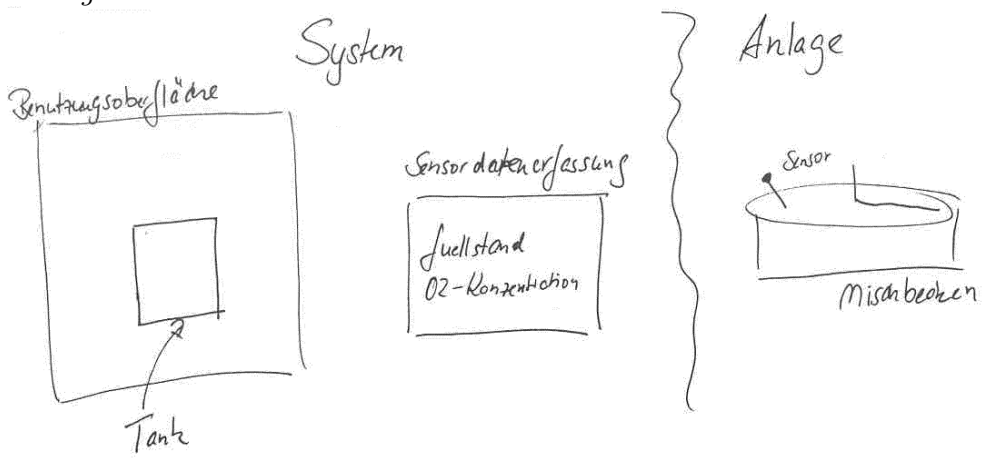


Abbildung 1.2: Entwurf der Elemente

[1]Dies wurde auch durch die Studie in Abschnitt 3.3.5 belegt.

Als nächstes wurden bestimmte Eigenschaften formuliert, z. B. daß der Tank füllbar ist, daß die Füllhöhe und die Farbe von den Sensordaten abhängen:

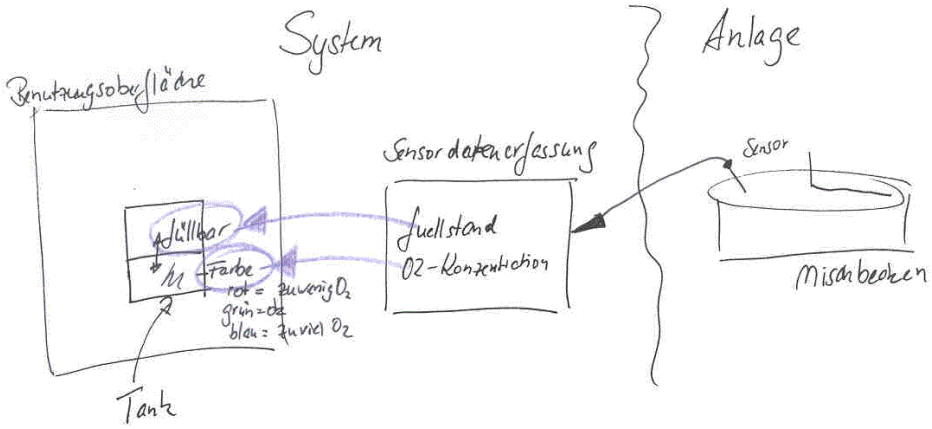
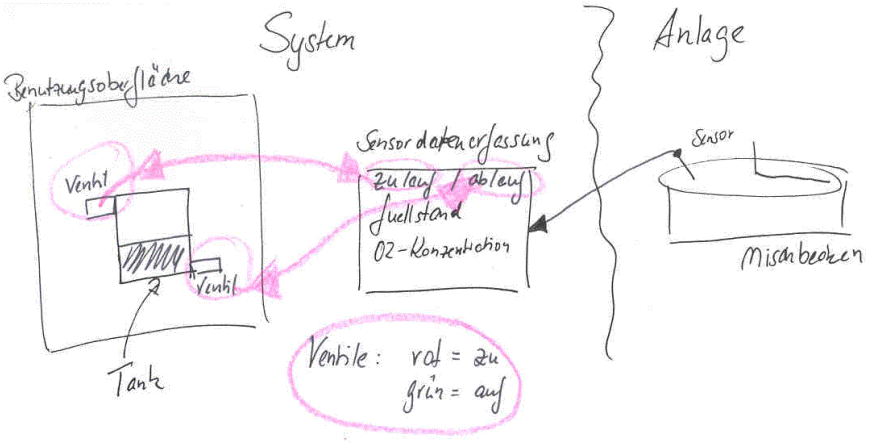


Abbildung 1.3: Verhalten der Oberflächenelemente

Spezifikation weiterer Elemente und ihrer Eigenschaften:



usw.

Abbildung 1.4: Abhängigkeiten

Beispiel 2: Benutzungsschnittstellen in Flugzeugen

Komplexere Benutzungsschnittstellen setzen sich oft aus mehreren graphischen Bildschirmelementen zusammen. Die Abbildungen 1.5 und 1.6 zeigen Benutzungsschnittstellen in Flugzeugen, die sich aus mehreren Einzelelementen zusammensetzen.

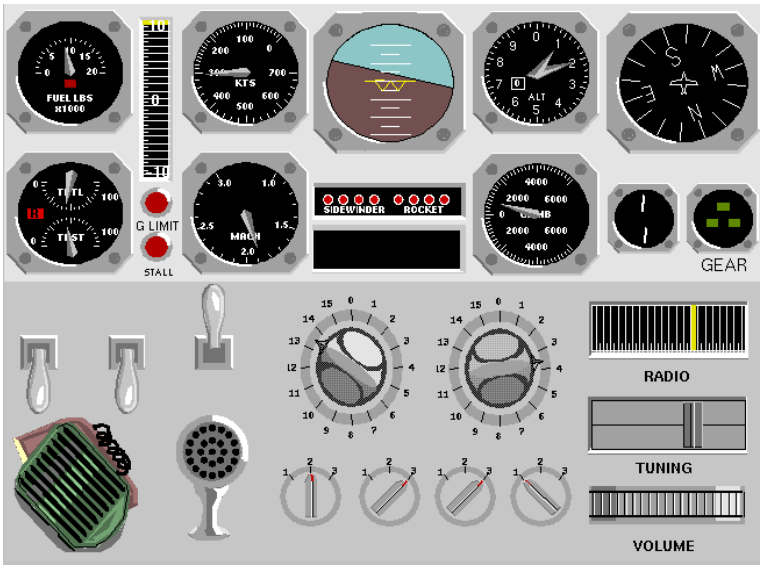


Abbildung 1.5: Ein klassisches Flugzeugdisplay. Dieser Bildschirm ist aus vordefinierten Bildschirmelementen der Firma Generic Logic Inc. [Gen00] zusammengestellt.

Neuere Flugzeugdisplays fassen die Daten mehrerer Sensoren zusammen und zeigen ein Bild, das diese Daten mit anderen Informationen integriert.

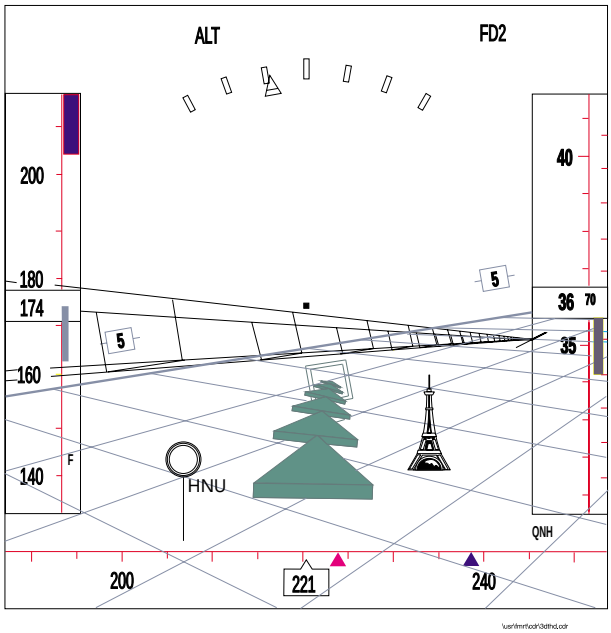


Abbildung 1.6: Ausschnitt aus einem Flugzeugdisplay für Blindflüge, z. B. bei Nebel (nach [Hel94]).

Bei genauerer Analyse der Abbildung 1.6 entdeckt man, daß auf einem Video-Hintergrundbild (oder Koordinatensystem) mehrere einfache Bildschirmelemente angeordnet sind: Durchlaufende Skalenelemente zeigen Geschwindigkeit und Flugrichtung an und eine einfache Skala die Höhe. Eine quer durch den Bildschirm laufende Linie gibt die Fluglage an. Die Dreiecke

in der Bildmitte zeigen kombiniert die Richtung und Lage für den momentanen Zeitpunkt (erstes Dreieck) und vorausberechnet für spätere Zeitpunkte an.

Alle Elemente stellen also Daten der Anwendung durch graphisches Verhalten dar.

Die Implementierung eines solchen Flugzeugdisplays „von Hand" hat sogar *vereinfacht* im Rahmen eines von mir betreuten studentischen Praktikums mehrere Monate gedauert [KPZ96]. Es wurde mit Hilfe des Software-Baukastens Amulet [MFM+96] realisiert, der flexibel und einfach zu bedienen ist.

1.2 Einordnung der Arbeit und Schwerpunkt

Der Entwicklungsprozeß von Benutzungsschnittstellen und die an der Entwicklung beteiligten Personen

Abbildung 1.7 zeigt einen Überblick über den Prozeß des Entwurfs von Benutzungsschnittstellen. Die verschiedenen Schritte werden mehrfach verfeinert, was durch die grauen Pfeile verdeutlicht wird.

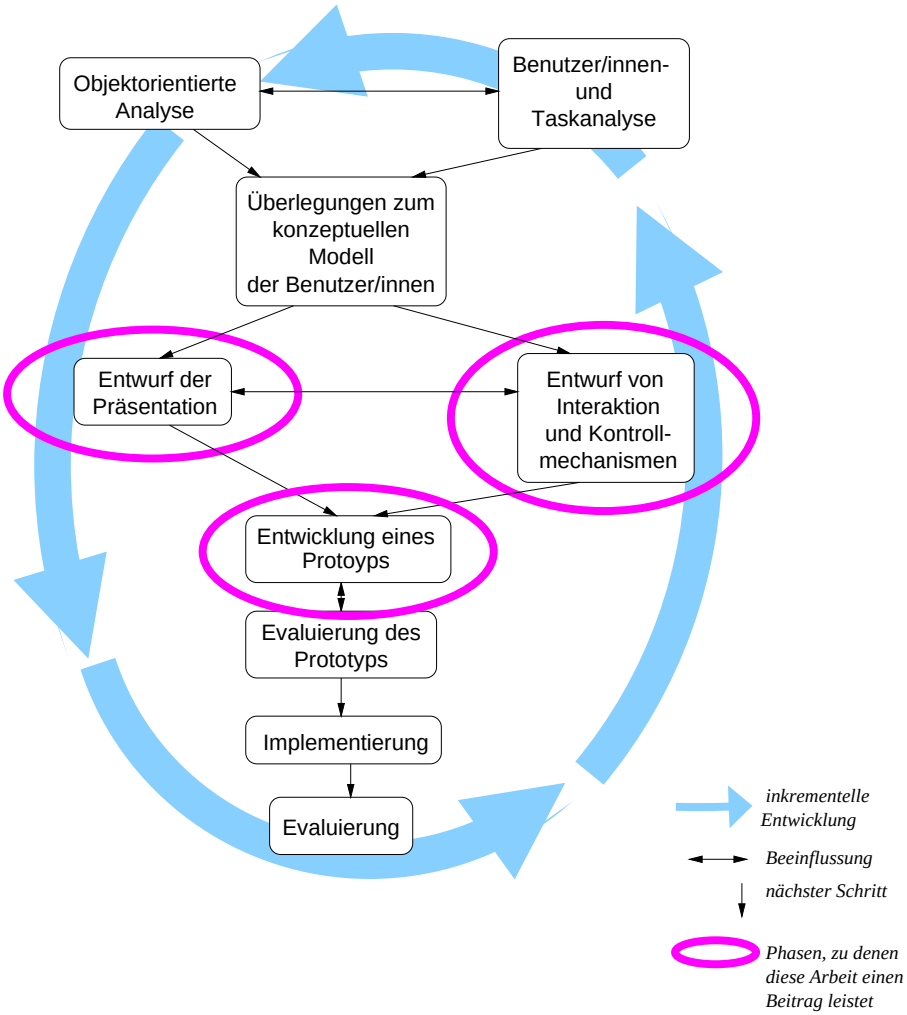


Abbildung 1.7: Der Prozeß der Realisierung einer objektorientierten Benutzungsschnittstelle (nach [Col94])

Für jeden Schritt des Entwurfsprozesses existieren Entwurfsmethoden, z. B. für die Aufgabenanalyse und die Analyse der Benutzer/innen oder den Entwurf des konzeptuellen Modells.

Einordnung dieser Arbeit in das vorgestellte Prozeßmodell

Die vorliegende Arbeit beschäftigt sich mit den Schritten zur Spezifikation der Benutzungsschnittstellen-*Software*, d. h. mit den Phasen

- „Entwurf der Präsentation",
- „Entwurf von Interaktions- und Kontrollmechanismen" und
- „Entwicklung des Prototyps".

Der Schwerpunkt liegt dabei auf visuellen Spezifikationsmethoden.

Zur Terminologie

An der Benutzung der Schnittstelle und diesen Entwicklungsschritten sind verschiedene Personen beteiligt.

- Die *(End-)Benutzer/innen* sind diejenigen, die mit einer Software-Anwendung mit Hilfe der Benutzungsschnittstelle arbeiten, z. B. Operateur/innen oder Expert/innen des Anwendungsgebiets.
- Die Benutzungsschnittstelle wiederum kann von unterschiedlichen Personen entwickelt werden, die in dieser Arbeit folgendermaßen bezeichnet werden:
  - *Entwickler/in* wird in dieser Arbeit diejenige Person genannt, die den *Software-Entwicklungsprozeß* in den verschiedenen Phasen (Objektorientierte Analyse, Design, Implementierung) durchführt. Um den Software-Entwicklungsprozeß von der Gesamtentwicklung (einschließlich Aufgaben- und Benutzer/innen-Analyse) abzugrenzen, wird an einigen Stellen auch der Begriff *Konstruktion* verwendet. Wenn sie nur implementiert, wird sie *Programmierer/in* genannt. Wenn sie nur die Analyse der Benutzer/innen und deren Aufgaben übernimmt, wird sie *Entwickler/in des Benutzer/innen-* oder *Aufgabenmodells* genannt.
  - Mit Designer/innen werden die Personen bezeichnet, die die graphische Gestaltung übernehmen[2].
  - Auch (End-)Benutzer/innen können Benutzungsschnittstellen entwickeln, wenn sie entweder
    - ebenfalls einer der anderen Kategorien zugeordnet werden können
    - oder ihnen entsprechende Werkzeuge zur Verfügung gestellt werden.

[2]Im Englischen wird der Begriff *designer* sowohl für Entwickler/innen als auch Designer/innen verwendet.

1.3 Aufbau dieser Arbeit

Die Struktur dieser Arbeit orientiert sich an dem beschriebenen und in Abbildung 1.8 gezeigten Lösungsweg.

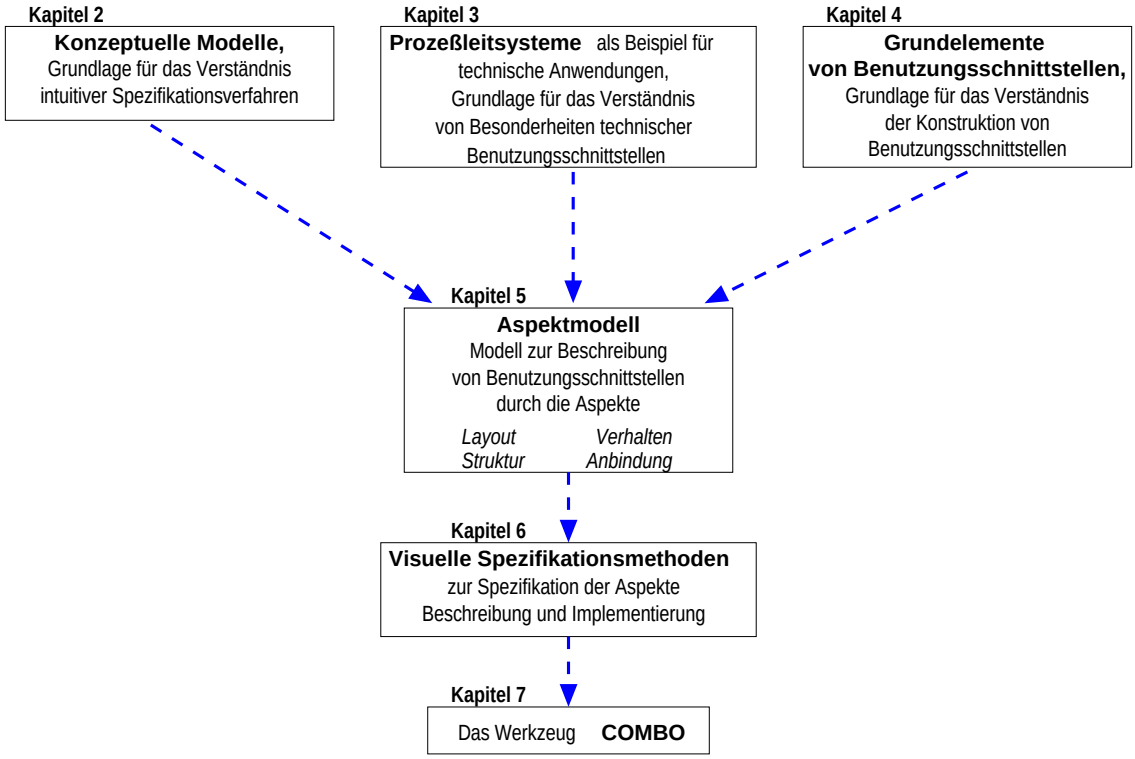


Abbildung 1.8: Entwurf von Benutzungsschnittstellen technischer Systeme mit visuellen Methoden und Werkzeugen

In **Kapitel 2** wird erforscht, welche *Vorstellungen* (das *sog. konzeptuelle Modell*) und *Fähigkeiten* Menschen haben, die Benutzungsschnittstellen entwickeln und programmieren (Abschnitt 2.1.1).

Daraus wird abgeleitet, welche Anforderungen an Entwurfsverfahren oder Werkzeuge zur Entwicklung von Benutzungsschnittstellen gestellt werden müssen, die das konzeptuelle Modell unterstützen (Abschnitt 2.2). Insbesondere wird untersucht, welche kognitiven Anforderungen *visuelle Entwurfsverfahren oder Werkzeuge* stellen (Abschnitt 2.2.6). Anhand einer empirischen Untersuchung wird gezeigt, wie sich *aufgabenorientierte visuelle Programmierung* von *visueller Programmierung der programmiertechnischen Mechanismen* (Abschnitt 2.2.6) unterscheidet, und daß sie auf *verschiedenen Programmierebenen* stattfindet.

Aus den gewonnenen Erkenntnissen werden *allgemeine Prinzipien* abgeleitet, auf denen visuelle Entwurfsmethoden beruhen müssen, wenn sie zur Unterstützung des Entwurfsprozesses eingesetzt werden sollen (Abschnitt 2.3).

Kapitel 3 identifiziert die besonderen Anforderungen *technischer Benutzungsschnittstellen* im Vergleich mit anderen Benutzungsschnittstellen, wie Büroanwendungen oder CAD-Systeme:

- Die Benutzer/innen haben andere Aufgaben (Beobachten, Bedienen, Steuern statt Entwerfen, Daten eingeben, ...), weshalb die Benutzungsschnittstellen anders aussehen, insbesondere mehr Graphikanteile enthalten müssen (Abschnitte 3.1 und 3.2).

- Diese Besonderheiten der technischen Benutzungsschnittstellen zusammen mit der Berücksichtigung von interdisziplinären Entwicklungsteams führen zu besonderen Anforderungen an den Entwicklungsprozeß und an die eingesetzten Entwurfsmethoden und Werkzeuge.

Anhand einer Studie (3.3.5) wird untersucht, wie sich diese Anforderungen auf die Entwicklung von Benutzungsschnittstellen auswirken.

Kapitel 4 widmet sich den programmiertechnischen Grundlagen für den Entwurf von Benutzungsschnittstellen. Um die gemeinsamen Grundlagen zu finden, werden im Rahmen dieser Arbeit verschiedene Ansätze zu folgenden Themenbereichen analysiert:

- Grundsätzlicher Aufbau von Benutzungsschnittstellen.

- Architekturen für Benutzungsschnittstellen.

- Mechanismen für interaktive Systeme in Klassen- und Funktionsbibliotheken sowie Sprachkonzepte.

- Einsatz von Software-Engineering-Konzepten wie Entwurfsmuster und Komponententechnologie bei der Entwicklung von Benutzungsschnittstellen.

- Visuelle Entwurfsmethoden und Werkzeuge aus den Gebieten *Software Engineering*, *objektorientierte Programmierung*, *visuelle Programmierung*, *Mensch-Maschine-Schnittstellen* (*HCI*), *Elektrotechnik* und *Maschinenbau.* Insbesondere wird der Einsatz und die Integration visueller Spezifikationsmethoden in Programmierumgebungen für die Sprachen Java und Smalltalk anhand der in Kapitel 2 genannten kognitiven Anforderungen an Entwickler/innen untersucht.

Es wird damit herausgearbeitet, welche programmiertechnischen Aufgaben beim Entwurf einer Benutzungsschnittstelle zu lösen sind.

Kapitel 5 führt das *Aspektmodell* als Modell für die Beschreibung der in Kapitel 4 herausgefundenen Aufgaben ein. Benutzungsschnittstellen werden durch die folgenden Aspekte beschrieben:

- *Layout* (graphische Anordnung),

- *Objektstruktur* (programmtechnischer Bau- oder Architekturplan),

- *Verhalten der Benutzungsschnittstelle* sowie

- *Verbindung von Benutzungsschnittstelle und Anwendung.*

Mit dem Aspektmodell lassen sich sowohl die identifizierten programmiertechnischen Grundprinzipien als auch die Besonderheiten von technischen Benutzungsschnittstellen beschreiben. Auch die in Kapitel 2 gefundenen Prinzipien können mit dem Modell auf allen Ebenen unterstützt werden.

Aufbauend auf diesem Modell wird in **Kapitel 6** zunächst eine Basisarchitektur für das Werkzeug COMBO vorgestellt. Daraus werden visuelle Spezifikationsmethoden entwickelt, die prototypisch implementiert und in COMBO integriert sind. Die grundlegende Idee dabei ist, daß COMBO verschiedene Entwurfsmethoden anbietet, die jeweils einen Teilaspekt der Benutzungsschnittstelle entwerfen.

In diesem Kapitel wird zu jedem Aspekt eine kurze Übersicht prinzipiell infrage kommender Methoden gegeben. Danach werden einzelne Methoden beispielhaft mit Hilfe der in den vorhergehenden Kapiteln erarbeiteten Kriterien (konzeptuelles Modell, Programmierebene, visuelle Prinzipien, programmiertechnische Grundprinzipien) genauer vorgestellt.

Damit wird der oben vorgestellte Prozeß der Entwicklung verfeinert, wie in Abbildung 1.9 dargestellt:

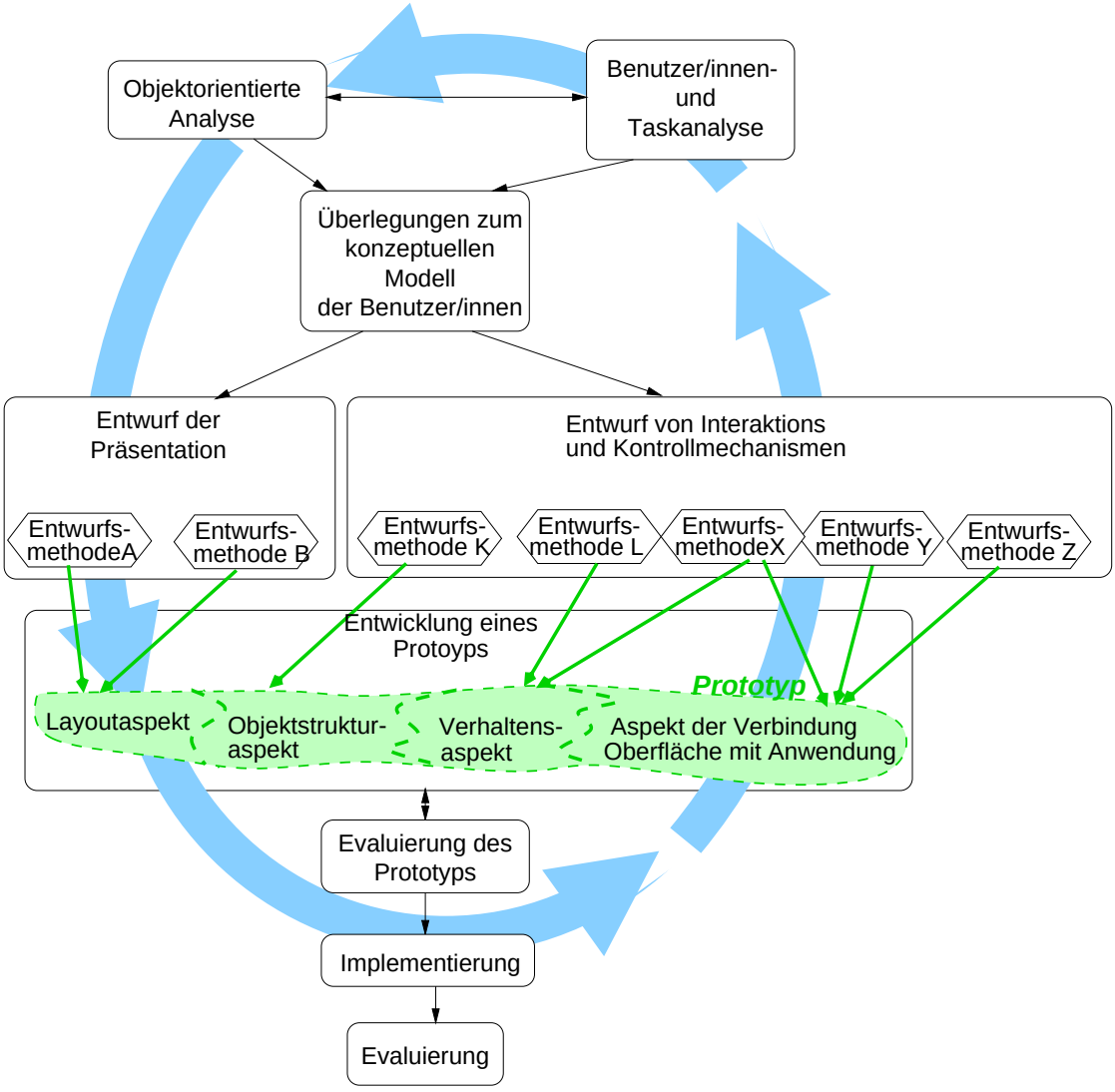


Abbildung 1.9: Einordnung der Ergebnisse der vorliegenden Arbeit in den Entwurfsprozeß

Kapitel 7 stellt die Benutzungsschnittstelle und die Implementierungsfragen des Werkzeugs COMBO vor. Es wird erläutert, wie die Zusammenarbeit der verschiedenen Arten von Editoren unterstützt wird.

Weiterhin wird exemplarisch gezeigt, wie ein Beispiel durchgehend entwickelt werden kann.

Dieses Beispiel wurde an einer Versuchsperson getestet und mit der herkömmlichen Art verglichen, das Beispiel zu entwickeln.

Schließlich werden einige Erweiterungen von COMBO vorgestellt und diskutiert.

Kapitel 8 faßt die Ergebnisse dieser Arbeit zusammen und gibt einen Ausblick auf mögliche zukünftige Forschungsarbeiten.

Kapitel 2

Wissen und Fähigkeiten von Entwickler/innen: Wie sich konzeptuelle Modelle bilden

"Software darf kein Handbuch brauchen"
"Software soll keine Hilfe brauchen"
"Software muß sich selbst erklären"
[Bis00]

Das Ziel dieser Arbeit ist die Unterstützung des Entwurfs von Benutzungsschnittstellen, d. h. Unterstützung für die Entwickler/in bzw. Programmierer/in. Dazu wird in diesem Kapitel zunächst (Abschnitt 2.1.1) gezeigt, welche konzeptuellen Modelle die *Benutzer/innen* haben, und welche konzeptuellen Modelle die *Entwickler/innen* von Benutzungsschnittstellen beeinflussen und welche sie berücksichtigen müssen.

Danach wird in Abschnitt 2.2 untersucht, welche grundlegenden Fähigkeiten für die verschiedenen Methoden (textuelles Programmieren, visuelles Entwerfen und Programmieren) beim Entwurf von Benutzungsschnittstellen benötigt werden. Daraus werden Grundprinzipien für das in dieser Arbeit entwickelte Werkzeug COMBO abgeleitet.

Konkrete Methoden, die durch COMBO umgesetzt werden, werden aus den folgenden Kapiteln über anwendungsspezifische Benutzungsschnittstellen, den Aufbau und die Funktion von Benutzungsschnittstellen abgeleitet.

2.1 Konzeptuelle Modelle von Benutzer/in und Entwickler/in als Grundlage für Entwurfsverfahren

2.1.1 Das konzeptuelle Modell der *Benutzer/in* über Softwaresysteme

Die Benutzer/in eines Softwaresystems gewinnt bei der Ausbildung, Einarbeitung und Benutzung eine Vorstellung über das System, die *konzeptuelles oder mentales Modell* genannt wird und ein Teil des deklarativen Benutzer/innenwissens ist [Wan93].

Das konkrete konzeptuelle Modell einer Benutzer/in hängt von allgemeinen und persönlichen Faktoren ab. *Persönliche Faktoren* sind z. B. Vorwissen, Lerntyp, Denktyp. Aufgrund dieser Faktoren entwickelt jede Benutzer/in ihr eigenes konzeptuelles Modell. Die *allgemeinen Faktoren*, z. B. verwendete Metapher oder Verhalten des Systems, finden sich in den konzeptuellen Modellen *aller* Personen, die das System benutzen. Daher erlauben sie Verallgemeinerungen und Annahmen über das konzeptuelle Modell der „Durchschnittsbenutzer/in", wie die Puppen in Automobiltests. Deshalb ist es wichtig, das konzeptuelle Modell bei der Entwicklung eines Softwaresystems zu planen und zu berücksichtigen (soweit es möglich ist).

Beim Entwurf eines Softwaresystems muß folgendes berücksichtigt werden:

- Die Benutzer/in soll ein klares Bild von der Funktionsweise aller wichtigen Funktionen bekommen und
- die wichtigen Funktionen sollen klar voneinander abgegrenzt erscheinen.

Ein Teil dieser Punkte wird durch die Art der Einarbeitung, d. h. die Dokumentation, Lehrbücher und Seminare beeinflußt, aber wie schon im einleitenden Zitat gesagt wird, soll Software möglichst selbsterklärend sein.

Entwicklung der Vorstellung im Laufe der Zeit

Das konzeptuelle Modell ändert sich bei der Benutzung, indem zusätzliche Information über das System einbezogen wird. Bei der Einarbeitung und anfänglichen Benutzung eines Softwaresystems spielen das Wissen über bekannte, ähnliche Systeme, Metaphern, die zur Darstellung benutzt werden, sowie Erklärungen aus Dokumentationen, Büchern und Seminaren eine Rolle. Im Laufe der Zeit wird das Softwaresystem selbst zum Referenzsystem zum Aufbau konzeptueller Modelle über andere Softwaresysteme.

Adaptive und adaptierbare Systeme

Damit ein Softwaresystem für die Benutzer/in *leicht benutzbar und erlernbar* wird, muß das konzeptuelle Modell, das die Entwickler/in über das von ihr zu entwickelnde System hat, mit dem der Benutzer/in übereinstimmen. Dazu gibt es zwei Ansätze (z. B. in [Thi94]):

1. Beim Entwickeln wird das Softwaresystem von vornherein an ein konkretes konzeptuelles Modell (z. B. von der Durchschnittsbenutzer/in) angepaßt. Solche Systeme passen sich zur Laufzeit nicht weiter an die Benutzer/in an.
2. Das System paßt sich im Laufe der Benutzung an das (sich verändernde) konzeptuelle Modell der Benutzer/in an durch:
   - Flexible Benutzungsstrategien, d. h. die Benutzer/in kann zwischen verschiedenen Strategien wählen, z. B. Menübenutzung oder Abkürzungen durch die Tastatur (adaptierbare Systeme).
   - Intelligente Anpassung, d. h. das System beobachtet die Entwickler/in, zieht Schlüsse aus ihrem Verhalten und ändert sich entsprechend (adaptive Systeme).

Die Wichtigkeit von Erklärungen erfahrener Benutzer/innen, Tutorien oder Bücher

Das konzeptuelle Modell entsteht im Unterbewußtsein/Gehirn der Benutzer/in und ist (heutzutage) Untersuchungen nicht direkt zugänglich.

Daher ist die Untersuchung von konzeptuellen Modellen indirekt auf die Aussagen und Darstellungen der Benutzer/innen beschränkt.

Ein Vorgehen zum Abfragen eines konzeptuellen Modells ist, das Softwaresystem von einer erfahrenen Benutzer/in erklären zu lassen und diese Erklärung als Abbild eines „guten" konzeptuellen Modells zu verwenden[1]. Dieses Abbild besteht aus einer Menge von *Verständnismustern* und Regeln über das Softwaresystem, die das Verhalten der Elemente (z. B. Funktionen, Prozesse oder Objekte) einzeln oder im Verhältnis zueinander erklären. Diese Elemente sind sowohl Elemente des Systems als auch die Elemente, die durch das System manipuliert werden (z. B. das Textverarbeitungssystem und die dadurch erzeugten Texte).

Solche gefundenen Verständnismuster können benutzt werden, um bestehende Softwaresysteme zu verbessern oder neue Systeme zu konstruieren, die der Benutzer/in gut angepaßt sind.

Für die Entwicklung von Standardsoftware können dabei nur die allgemeinen Faktoren des konzeptuellen Modells übernommen werden.

Im folgenden wird in diesem Kapitel der Begriff konzeptuelles Modell im Anwendungsfeld „Entwurf von Benutzungsschnittstellen" weiter verfeinert.

2.1.2 Das konzeptuelle Modell der *Entwickler/in* von Benutzungsschnittstellen

Im vorhergehenden Abschnitt wurde das konzeptuelle Modell der Benutzer/in vorgestellt.

Das Ziel dieser Arbeit ist die Unterstützung des *Entwicklungsprozesses* von Benutzungsschnittstellen. Daher wird nun untersucht, welches konzeptuelle Modell die *Entwickler/in einer Benutzungsschnittstelle* hat, im Gegensatz zum konzeptuellen Modell der *Endbenutzer/in*, d. h. derjenigen Person, die die produzierten Benutzungsschnittstellen benutzen wird.

Natürlich muß die Entwickler/in einer Benutzungsschnittstelle auch das konzeptuelle Modell der Endbenutzer/in berücksichtigen.

Dieses konzeptuelle Modell der Entwickler/in muß in einem Werkzeug, das den Entwurf von Benutzungsschnittstellen unterstützt, berücksichtigt werden.

Das konzeptuelle Modell der ***Entwickler/in von Werkzeugen zur Unterstützung der Entwicklung von Benutzungsschnittstellen*** (z. B. der Autorin dieser Arbeit) beruht daher auf dem konzeptuellen Modell der ***Entwickler/in von Benutzungsschnittstellen***, das, wie gesagt, das konzeptuelle Modell der ***Benutzer/in der Benutzungsschnittstelle*** umfaßt.

[1] Andere Methoden sind: Interviews, Fragebögen, statistische Auswertung von Bewertungen, Computersimulationen [RW92] und Modellierung von Ähnlichkeitsdaten [Wan93]

Diese komplizierte Beziehung zeigt Abbildung 2.1 für den Fall des Tank-Beispiels.

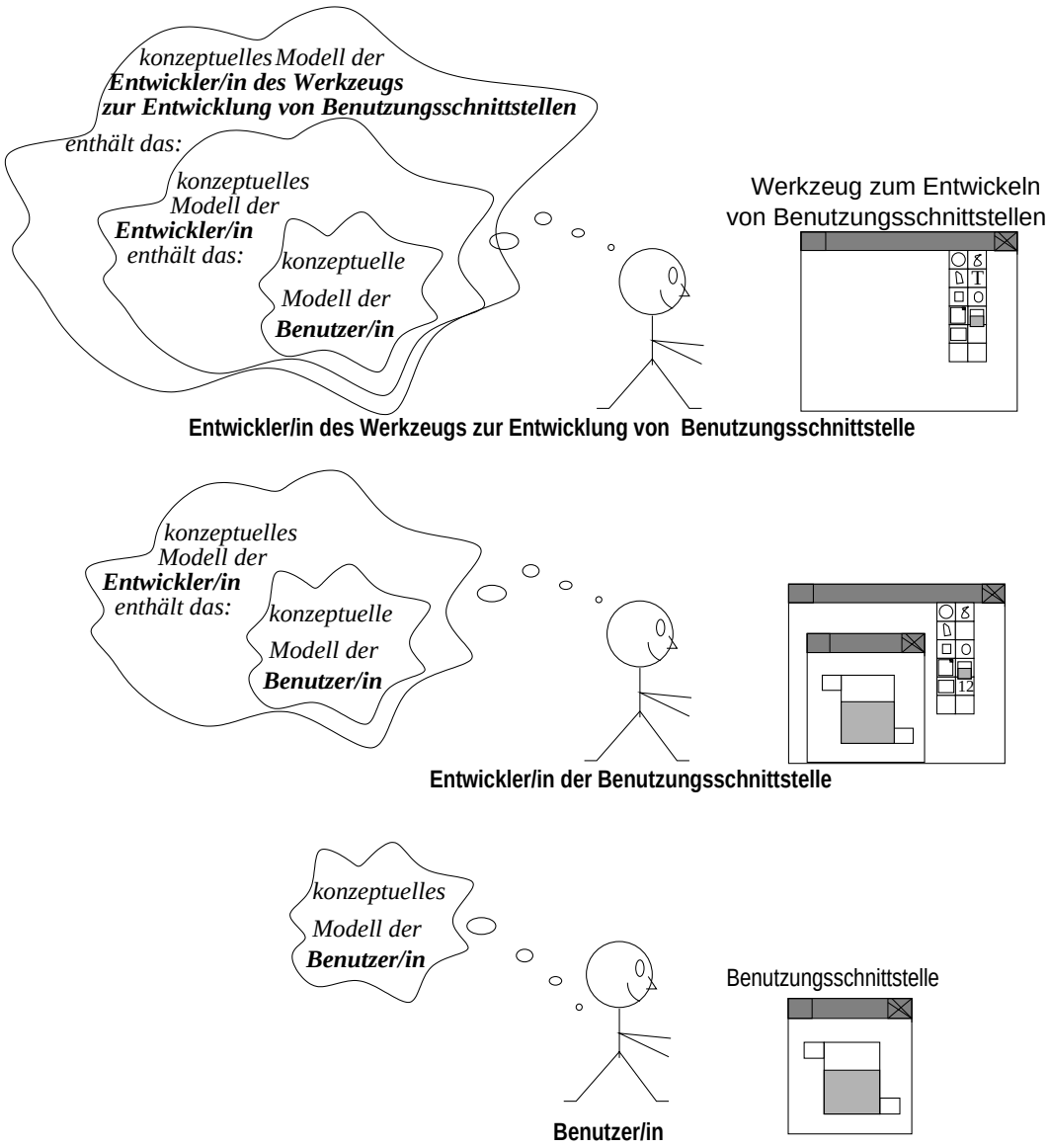


Abbildung 2.1: Die Schachtelung konzeptueller Modelle bei den an der Benutzung und Entwicklung von Benutzungsschnittstellen beteiligten Personen

Die konzeptuellen Modelle der einzelnen Personen setzen sich wiederum aus Vorstellungen über die verschiedenartigen psychologischen und technischen Modelle zusammen (siehe z. B. [ZZ92][Vää95]). Abbildung 2.2 zeigt eine Übersicht über die in der Literatur genannten Modelle, wiederum am Beispiel des Tanks, danach werden sie kurz beschrieben.

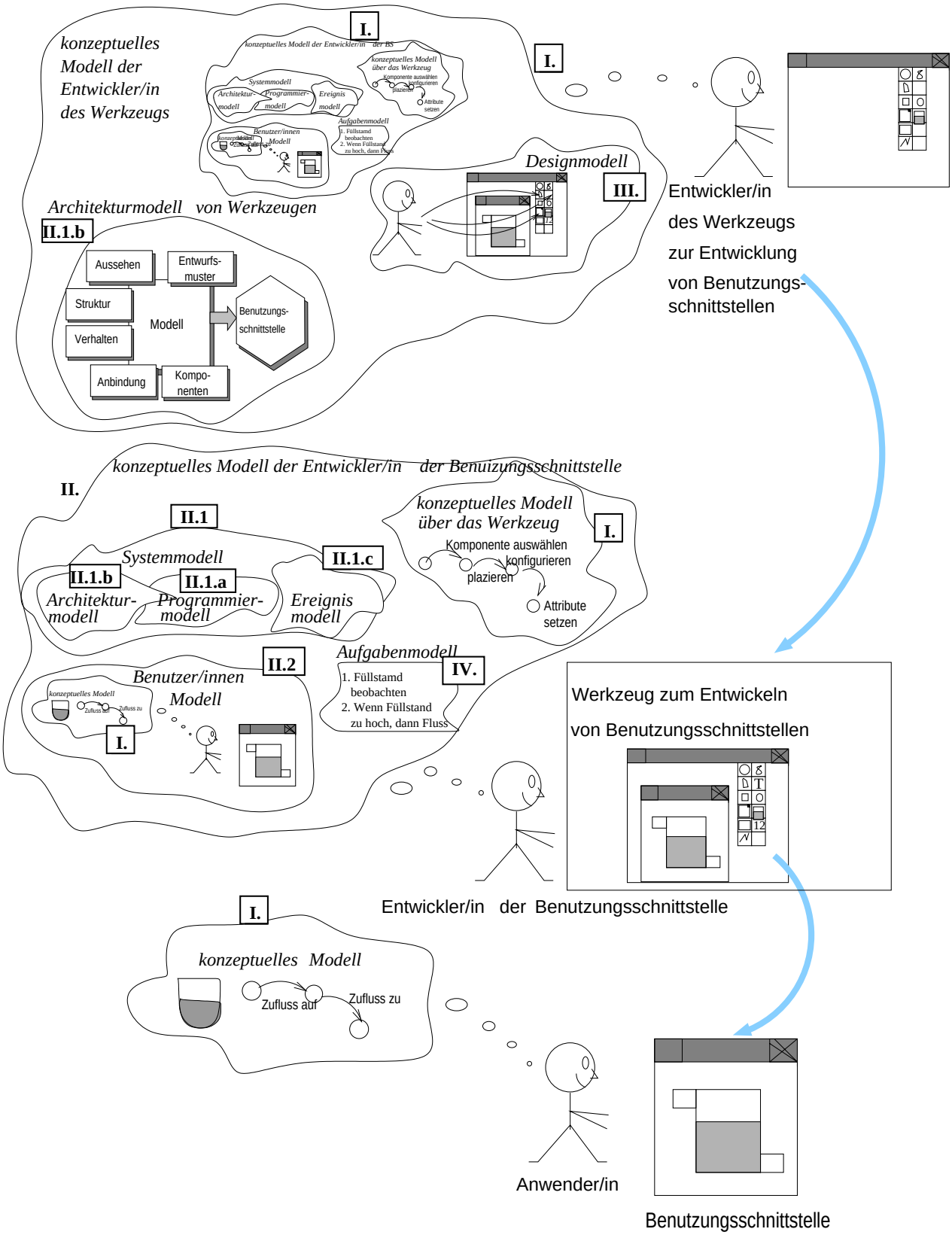


Abbildung 2.2: Die verschiedenen konzeptuellen Modelle

Im folgenden werden die konzeptuellen Modelle im einzelnen vorgestellt, um zu verdeutlichen, wie viele Modelle die Entwickler/in einer Benutzungsschnittstelle kennen muß.

I. Das konzeptuelle Modell der Benutzer/in enthält die Vorstellung der Benutzer/in über die Benutzungsschnittstelle sowie das dahinterliegende Anwendungssystem.

II. Das Entwurfsmodell ist der Oberbegriff für konzeptuelle Modelle der Entwickler/innen über den Entwurf. Es enthält das Systemmodell und das Benutzer/innenmodell.

II.1 Das Systemmodell ist das konzeptuelle Modell der Entwickler/innen über die Entwicklungsumgebung, mit der die Benutzungsschnittstelle programmiert wird. Die Entwicklungsumgebung enthält hier relevant das Programmiermodell, das Ereignismodell und das Architekturmodell.

II.1.a Das Programmiermodell (auch program model [IBM97a]**)** ist das Paradigma, bzw. die konkreten Mechanismen, nach denen die Programmierung erfolgt. Dieses Modell ist ein echtes Systemmodell, aber gleichzeitig ein Teil des konzeptuellen Modells der Entwickler/innen .

II.1.b Das Ereignismodell beschreibt einen speziellen Ausschnitt des Programmiermodells, nämlich den Teil, der erläutert, wie Ereignisse (z. B. Benutzer/inneneingaben) verarbeitet werden.

II.1.c Das Architekturmodell beschreibt die Struktur und das grundlegende Verhalten der zu entwickelnden Benutzungsschnittstelle. Ebenso wie das Programmiermodell handelt es sich hier um ein Systemmodell; einige mögliche Architekturmodelle werden in Kapitel 4.1 vorgestellt, und die Vorstellung darüber ist wieder ein Teil des konzeptuellen Modells der Entwickler/in.

Für das konzeptuelle Modell der Benutzer/in der Benutzungsschnittstelle sind Programmier-, Ereignis- und Architekturmodell transparent. Allerdings beeinflußt die Wahl der Sprache/Klassenbibliothek unter Umständen Aussehen und Möglichkeiten der fertigen Benutzungsschnittstellen, z. B. sehen die graphischen Elemente von Bentuzungsschnittstellen, die sog. graphischen Bausteine oder *Widgets,* je nach Bibliothek verschieden aus.

II.2 Das Benutzer/innenmodell beschreibt die Vorstellung, die die Entwickler/in von der künftigen Benutzer/in hat. Dieses Modell schließt sowohl allgemeine Daten der voraussichtlichen Durchschnittsbenutzer/in (Alter, Bildungsgrad etc.), als auch das vermutete konzeptuelle Modell der Benutzer/in über die zu entwickelnde Benutzungsschnittstelle ein. Für Standardsoftware werden in der Regel Benutzer/innenklassen festgelegt, die durch spezielle Dialog- und Interaktionstechniken unterstützt werden können.

Die bisher vorgestellten Modelle spiegeln die *technische Sicht* der Programmentwicklung wider. Eine andere Sicht ist die sog. *Benutzer/innenzentrierte Sicht*, die ebenfalls konzeptuelle Modelle benennt. Hammond et al. [HGCM87] nennen diese beiden Sichten *functional specification* und *usability specification.*

III. Designmodell Ein solches Modell ist das sog. **design model** (dts. Designmodell)[IBM97a]. Dieses Modell beschreibt die Absicht, die die Entwickler/in in Bezug auf die Arbeitsabläufe der Benutzer/in bei der Auswahl der Elemente verfolgt. D. h. die Arbeitsabläufe zur Erledigung von Aufgaben werden, durch Auswahl geeigneter graphischer Benutzungsschnittstellen-Elemente, in möglichst intuitive Schritte bei der Benutzung übersetzt. Die Ähnlichkeit von Designmodell und bei der späteren Benutzung gefundenem konzeptuellem Modell der Benutzer/in ist ein Maß dafür, wie intuitiv eine Benutzungsschnittstelle ist. Das Designmodell ist von zentraler Bedeutung, wenn es um die Gestaltung der Benutzungsschnittstelle geht. Da in dieser Arbeit der Schwerpunkt unabhängig von ihrer Güte auf der *Konstruktion* der Benutzungsschnittstelle liegt, wird das *Designmodell der zu entwickelnden Benutzungsschnittstelle* hier nicht weiter untersucht. Statt dessen steht das *Designmodell des zu entwickelnden Werkzeugs* zur Unterstützung der Konstruktion im Mittelpunkt.

IV. Aufgabenmodell Das Aufgabenmodell (engl. ***task model***) beschreibt, welche Aufgaben die Benutzer/in hat, um daraus die Anforderungen für die Benutzungsschnittstelle abzuleiten und zu gewährleisten, daß die Aufgaben ebenfalls mit Hilfe des Softwaresystems gelöst werden können. Die Erstellung des Aufgabenmodells erfolgt in einer früheren Phase als die in dieser Arbeit betrachteten Phasen und wird deshalb ebenfalls nicht weiter behandelt.

2.1.3 Konzeptuelle Modelle und ihr Einfluß für Entwurfswerkzeuge und -verfahren: Der *Erklärungsansatz*

Im Mittelpunkt der Betrachtungen dieser Arbeit steht das *konzeptuelle Modell der Entwickler/in der Benutzungsschnittstelle*. Besonders die Vorstellungen über das Systemmodell (II.1), bestehend aus Programmier-, Ereignis- und Architekturmodell sind von besonderem Interesse. Dieses Modell beschreibt Klassenbibliothek und Programmiersprache der Entwicklungsumgebung und damit wichtige Mechanismen und Strukturen, sowie Software Engineering-Konzepte, die zur vollständigen Programmierung der Benutzungsschnittstelle notwendig sind. Ebenso von Interesse sind die *konzeptuellen Modelle über die Benutzungsschnittstellen der Werkzeuge zum Entwurf von Benutzungsschnittstellen*.

Konkret soll als Ziel dieser Arbeit eine Vorgehensweise, basierend auf der Kombination mehrerer (visueller) Spezifikationsmethoden (und ein darauf aufbauendes Werkzeug), entwickelt werden, die die kognitiven Fähigkeiten der Entwickler/innen einbezieht und klare konzeptuelle Modelle erzeugt. Um dieses Ziel zu erreichen, wurde das Wissen über die Wichtigkeit von Erklärungen beim Aufbau konzeptueller Modelle genutzt: Aus der Art, wie erfahrene Entwickler/innen die Entwicklung von Benutzungsschnittstellen erklären, wurde ihr konzeptuelles Modell abgeleitet, und die dabei verwendeten Vorgehensweisen wurden in die Entwurfsmethode einbezogen. Dieses Vorgehen wird im Folgenden ***Erklärungsansatz*** genannt.

Daher sind die Erklärungen erfahrener Entwickler/innen über die Mechanismen der Umgebung, Sprache und Klassenbibliothek von Bedeutung.

Ein Beispiel:

In Lehrbüchern findet sich zur Erklärung des in modernen Programmierumgebungen (z. B. Smalltalk und Java) häufig verwendeten Model View Controller-Mechanismus (MVC-Mechanismus), eine Darstellung wie in Abbildung 2.3:

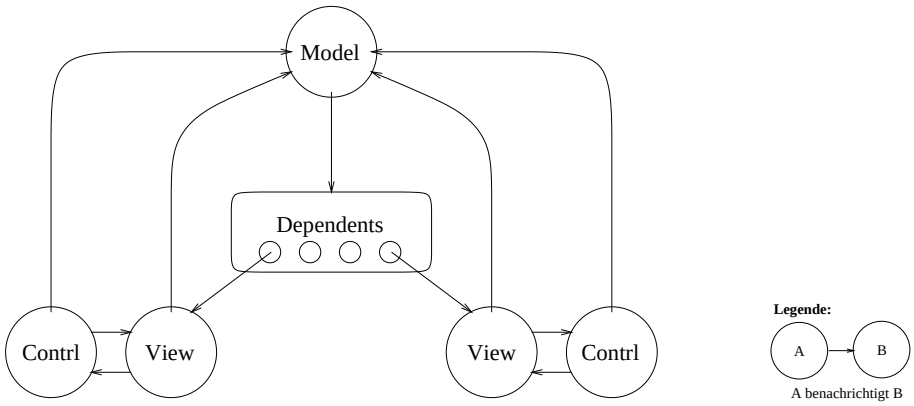


Abbildung 2.3: Beziehungen zwischen den beteiligten Klassen im MVC-Mechanismus

Solche Bilder werden von Expert/innen benutzt, um den MVC-Mechanismus zu erklären und prägen damit das konzeptuelle Modell der Entwickler/innen, die diesen Ansatz lernen. Je länger den Lernenden das Modell bekannt ist und von ihnen verstanden wird, desto mehr ist zu beobachten, daß sie selbst diese Bilder zum Erklären verwenden. Daher ist davon auszugehen, daß die Bilder das konzeptuelle Modell geprägt haben.

Solche Bilder erklären in der Regel aber nur einen Ausschnitt, einen (Teil-)Aspekt eines Modells, in diesem Fall einen Teil des Verhaltens der Benutzungsschnittstelle, und zwar derjenige, der durch den MVC-Mechanismus beschrieben wird. Das konzeptuelle Modell aller Aspekte einer Benutzungsschnittstelle wird in der Regel in vielen solcher Ausschnitte sowie mit Hilfe vieler Darstellungen gelernt, die jeweils einen Teilaspekt beschreiben.

Konkret fließen die konzeptuellen Modelle über die später in Kapitel 4 vorgestellten Elemente, Vorgehen und Mechanismen in den Entwicklungsprozeß von Benutzungsschnittstellen ein, das sind:

- Technischer Aufbau von Benutzungsschnittstellen auf der Ebene der Implementierung (Klassenbibliothek, Programmiersprache, Graphik- bzw. Fenstersystem).
- Architekturmodelle von Benutzungsschnittstellen.
- Methoden zum Entwurf von Benutzungsschnittstellen (HCI-Methoden).
- Methoden zum Entwurf von Softwaresystemen, also Methoden des Software Engineering
  - Objektorientierte Methoden
  - Software-Architekturen.
- Außerdem spielen auch die Entwurfsmethoden[2] der Anwendungsgebiete eine Rolle, weil die Anwendungsexpert/innen oft auch die Entwickler/innen der Benutzungsschnittstellen sind. Diese werden in Kapitel 3 beschrieben.

Das Verstehen der genannten Punkte und ihrer bildlichen Darstellung ist wichtig, um den Erklärungsansatz beim *Entwurf eines Werkzeugs zur Entwicklung von Benutzungsschnittstellen* verwenden zu können. Es muß nach Erklärungsgegenständen und -methoden gesucht

[2] zum Entwurf von Systemen des Anwendungsgebiets

werden, um eine flexible Werkzeugunterstützung gewährleisten zu können, denn jedes *konzeptuelle Modell von Strukturen und Mechanismen* von Benutzungsschnittstellen, sei es der Aufbau aus Objekten im Sinne objektorientierter Programmierung oder die Repräsentation durch Dialogsequenzen, impliziert eigene *konzeptuelle Modelle über die Vorgehensweisen bei der Benutzungsschnittstellen-Entwicklung.* Zu beachten ist außerdem, daß im Laufe der Entwicklung einer Benutzungsschnittstelle in den verschiedenen Ebenen (siehe Abschnitt 2.2.4) verschiedene Modelle zur Spezifikation benutzt werden, die aufeinander abgebildet werden müssen. Darüberhinaus werden sogar in einzelnen Phasen mehrere Modelle verwendet, was bei den Entwickler/innen zu Verwirrung führen kann.

2.1.4 Ein Vorgehen, das das konzeptuelle Modell der Entwickler/in berücksichtigt: Der *Aspektansatz.*

Die in dieser Arbeit vorgeschlagene Vorgehensweise und das unterstützende Werkzeug (siehe Kapitel 7) berücksichtigen die angesprochenen Punkte folgendermaßen:

- Ausgehend von in diesem Abschnitt vorgestellten konzeptuellen Modellen und den benötigten kognitiven Fähigkeiten beim Entwurf von Benutzungsschnittstellen (Entwerfen, Programmieren, visuelles Programmieren, im nächsten Abschnitt (2.2) vorgestellt), werden am Ende dieses Kapitels Grundprinzipien für Methoden und Werkzeuge abgeleitet.

- Zur Überprüfung dieser Grundprinzipien wurden bei Entwickler/innen von Benutzungsschnittstellen für technische Systeme in einer Studie die Erklärungen über Aufbau und Funktion von Benutzungsschnittstellen beobachtet. Siehe Kapitel 3.3.5.

- Darauf aufbauend wurde ein Modell über Benutzungsschnittstellen entwickelt (in Kapitel 5), indem verschiedene Aspekte herausgefiltert wurden, die Gegenstand der Erklärungen sind. Die These ist:

  > *Wenn die Benutzungsschnittstelle mit Hilfe dieser Aspekte erklärt werden kann, dann kann umgekehrt die Benutzungsschnittstelle auch aus Spezifikationen, die die Form solcher Erklärungen haben, konstruiert werden.*

  Dieser Ansatz wird im folgenden ***Aspektansatz*** genannt.

- Für jeden Aspekt wurde untersucht, mit welchen Methoden er erklärt und spezifiziert werden kann. Dabei werden sowohl bekannte Methoden herangezogen als auch im Rahmen dieser Arbeit eigene Methoden entwickelt. Siehe Kapitel 6.

- Die Entwickler/in kann sich dann aus einer Vielzahl von Erklärungs-/Spezifikationsmethoden die heraussuchen, mit denen sie am besten umgehen kann. Siehe Kapitel 5 und 7.

2.2 Kognitive Anforderungen beim Entwurf von Benutzungsschnittstellen

Unter *Kognitionswissenschaften* im weiteren Sinne werden alle Disziplinen bezeichnet, die ein Interesse an der Erforschung von Wissen und der Wissensverarbeitung haben, z. B. die sog. Künstliche Intelligenz, Linguistik und Kognitionspsychologie. Im engeren Sinne ist Informationsverarbeitung, die auf den Fähigkeiten und gehirnbiologischen Grundlagen von Menschen aufbaut, gemeint.

Um über den Erklärungsansatz hinaus die konzeptuellen Modelle der Entwickler/innen von Benutzungsschnittstellen beschreiben zu können, müssen die kognitiven Fähigkeiten der Entwickler/innen berücksichtigt werden. Beim Entwurf von Benutzungsschnittstellen werden Fähigkeiten zur Softwareentwicklung allgemein, zum visuellen Programmieren sowie speziell zum visuellen Programmieren von Benutzungsschnittstellen benötigt. Damit können insbesondere Strukturen und Mechanismen von Entwicklungsumgebungen von der kognitiven Seite her beschrieben werden. Die in diesem Kapitel beschriebenen kognitiven Grundlagen dienen auch als Basis für die drei Untersuchungen dieser Arbeit:

Untersuchung 1: Direktmanipulatives, aufgabenbezogenes Programmieren vs. struktur- und mechanismenorientiertes Programmieren anhand der Programmierung eines Roboters (Abschnitt 2.2.6)

Untersuchung 2: Konzeptuelle Modelle von Informatiker/innen und Ingenieur/innen über technische Benutzungsschnittstellen (Abschnitt3.3.5)

Untersuchung 3: Einsatz visueller Spezifikationsmethoden in Entwicklungsumgebungen für Smalltalk und Java (Abschnitt 4.5.2)

2.2.1 Kognitive Fähigkeiten, die bei der Software-Entwicklung benötigt werden

Fix et al. [FWS93] haben untersucht, was Anfänger/innen von Expert/innen beim Programmieren (von PASCAL-Programmen) unterscheidet, und fanden fünf Unterschiede im konzeptuellen Modell:

- Expert/innen haben ein hierarchisches und mehrschichtiges konzeptuelles Modell,
- sie können die Elemente verschiedener Schichten aufeinander abbilden,
- das konzeptuelle Modell gründet sich auf das Erkennen von Grundmustern,
- die Vorstellungen über die verschiedenen Elemente sind stark miteinander verknüpft und
- es gründet sich auf die Architektur- und Programmodelle.

Die Vorstellung über die Funktionsweise eines Programms entsteht aus der Vorstellung über die Ausführungsreihenfolge konkreter Programmteile [NC87] und der Interaktion von einzelnen Elementen [FWS93].

Denken in Umwegen

Eine weitere Fähigkeit, die bei der Benutzung existierender Programmierumgebungen vonnöten ist, ist das Beschreiten von Umwegen, um ein Ziel zu erreichen. Oft entsteht folgende Situation: Ein Lösungsansatz soll detailliert in der Programmiersprache formuliert werden. Viele Programmiersprachen (oder Bibliotheken) erlauben aber nicht die Formulierung der direkten Lösung, z. B. daß der Tank gefüllt wird. Deshalb muß die Entwickler/in lernen, in den Umwegen zu denken, die das Verhalten hervorruft, das sie erreichen will.

Eine solche Situation findet sich im Tank-Beispiel. Die visuelle Repräsentation besteht, graphisch formuliert, aus zwei Rechtecken, eines farbig, wobei es eine Relation zwischen der Größe des zweiten Rechtecks und dem Füllstand des Tanks gibt.

Ein solches Umwegdenken kann durch verschiedene Strategien vermieden werden, die in dieser Arbeit berücksichtigt wurden:

- Verwenden einer Kombination von Methoden, um in einer Situation, in der ein Formulieren der Lösung in einer Methode nicht möglich ist, auf eine andere zu wechseln.
- Anbieten von aufgabenbezogenen Methoden, die „natürliche" Ausdrücke sind. Im Tank-Beispiel wäre eine Methode hilfreich, bei der das Aussehen des Tanks gezeichnet werden kann und das Verhalten („füllbar") für dieses Element ausgewählt werden kann. Eine solche Methode wird in dieser Arbeit durch die Verhaltensbibliothek (siehe Abschnitt 6.3.1) angeboten.

2.2.2 Kognitive Fähigkeiten, die beim visuellen Programmieren benötigt werden

Visuelles Programmieren beruht in der Regel auf der Manipulation von Diagrammen. Aaron Sloman beschreibt in [Slo98], warum Diagramme so hilfreich sind. Er untersucht, in welchen Strukturen Menschen denken, schließen und handeln. Bisher ist es nicht möglich, die räumlichen und visuellen Fähigkeiten zu erklären, geschweige denn nachzuahmen. Anhand von zwei Beispielen, nämlich erstens der Frage, wieviele Möglichkeiten es gibt, die Unterhose auszuziehen, ohne die Jeans auszuziehen, und zweitens der Darstellung von unendlichen diskreten Mengen, zeigt er auf, daß weder die visuelle Präsentation von topologischen Abstraktionen noch die rein mathematische Abstraktion ausreichen, um die Lösungen schnell zu finden. Die Kombination aus abstraktem räumlichen und abstraktem logischen Schließen führt zu den besten Ergebnissen, ebenso wie in der Informatik Datenstrukturen zusammen mit den entsprechenden Funktionen erst die Gesamtfunktionalität eines Systems garantieren.

Visualisierung selbst bildet nicht unbedingt eine räumliche Struktur ab, sondern unsere räumliche Vorstellung von einer Struktur.

Es gibt zwei Arten von abstraktem räumlichen Schließen:

- Schlüsse aus einer Sequenz von räumlichen Strukturen ziehen und
- mögliche Manipulationen der Diagramme (nur die räumliche Lage der Elemente betreffend) betrachten und aus diesen Möglichkeiten Schlüsse ableiten.

Nach Sloman wird die zweite Möglichkeit von modernen visuellen Programmsystemen unterstützt. Die Menge von Struktur-Manipulationen (und Darstellungselementen) wird allgemein als Syntax der Darstellung bezeichnet. Daneben setzt sich die Semantik der diagrammatischen Darstellung daraus zusammen, welche andere interne oder externe Struktur, welche Aktionen oder Ziele die gegebene Struktur beschreibt, plant oder zusammenfaßt. Sloman vermutet, daß eine weitreichende Untersuchung nach Arten von Strukturmanipulationen Schlüsse auf die Denkweise des Gehirns zuläßt.

Für den Entwurf visueller Methoden in Diagrammform kann man folgende (sehr abstrakte) Konsequenzen ziehen. Wenn es eine Fähigkeit des Gehirns ist, Schlüsse aus diagrammatischen Repräsentationen zu ziehen, dann beschreiben diese Diagramme das dargestellte System, durch die direkte Darstellung und alle Schlüsse, die aus dem Diagramm gezogen werden können. Deshalb muß es auch möglich sein, Schlüsse in einem Diagramm zu kodieren, d. h. nicht alle Informationen in direkter Darstellung zu kodieren. Somit enthält die Aufgabe, ein Softwaresystem in Form von Diagrammen zu repräsentieren auch die Aufgabe, alle Informationen über das System und damit alle Schlüsse ebenfalls in das Diagramm einzubinden. Diagramme werden bei der Umsetzung von Strukturen und Mechanismen in abstrakte Darstellungen eingesetzt. Solche Diagramme spezifizieren methodisch genau diese Strukturen und Mechanismen, verbergen diese aber oft gegenüber der Entwickler/in (sog. higher-level concepts, siehe Abschnitt 2.2.6). Das Verbergen von Struktur wird dort untersucht. Beim Entwurf der Diagramme, die im Rahmen dieser Arbeit verwendet werden, muß berücksichtigt werden, daß der Computer aus den Informationen, die im Diagramm enthalten sind, die Informationen ableiten kann, die er braucht, um Software zu kodieren. Dies wird bei der Vorstellung der einzelnen Methoden in Kapitel 6 für die jeweilige Methode geklärt.

2.2.3 Kognitive Fähigkeiten, die bei der Entwicklung von Benutzungsschnittstellen mit visuellen Spezifikationsmethoden benötigt werden

Überwinden von Lernhürden

Brad Myers hat in einem eingeladenen Vortrag bei der VL 2000 (IEEE Symposium für visuelle Programmiersprachen) sowie in [Mye00a] das Problem der Lernhürden bei Benutzungsschnittstellen-Entwicklungswerkzeugen angesprochen: Je anspruchsvoller eine Anwendung ist, desto tiefer muß das Verständnis der Entwickler/innen über die Programmiermechanismen sein. Dieses Verständnis baut sich nicht immer linear auf (d. h. ausgehend von dem, was eine Programmierer/in bereits beherrscht, geht es in kleinen Schritten weiter zu mehr Verständnis), sondern oft muß man sehr viel Wissen ansammeln, um eine wenig kompliziertere Anwendung zu erzeugen.

Für einige Werkzeuge zeigt Abbildung 2.4 (aus [Mye00b]) solche Barrieren.

Wie in Abbildung 2.4 zu sehen ist, treten diese Hürden immer dann auf, wenn eine Programmiersprache, d. h. die konkreten Programmiermechanismen, zu lernen sind. Gerade bei der Entwicklung von Benutzungsschnittstellen können diese Hürden weiter in

1. programmiersprachlichen Hürden differenziert werden und
2. in Hürden, die sich aus dem Erlernen von Mechanismen und Strukturen der verwendeten Bibliotheken und Programmierumgebungen ergeben.

Eine Unterscheidung der Anforderungen in verschiedene Ebenen der Programmierung, die diese beiden Arten von Lernhürden berücksichtigt, wird im nächsten Abschnitt vorgestellt.

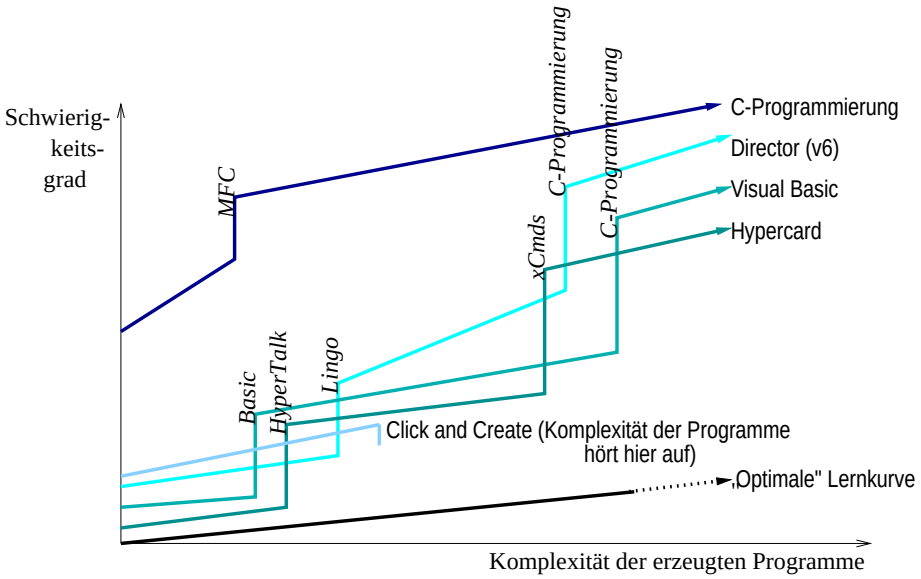


Abbildung 2.4: Lernkurven beim Erlernen verschiedener Programmiermethoden (nach [Mye00a])

2.2.4 Einordnung der beschriebenen Fähigkeiten in vier Ebenen der Programmierung

Die Komplexität von Aufgaben bei der Programmierung wird nach ihren Zerlegungsmöglichkeiten in verschiedene Stufen oder Ebenen unterschieden, z. B. werden nach [McK91] für die Roboterprogrammierung fünf Ebenen unterschieden: Aufgaben-, Handlungs-, Roboter-, Gelenk- und physikalische Ebene. Eine solche Unterscheidung ermöglicht eine klare Terminologie und führt damit zu einem strukturierten konzeptuellen Modell. Daher ist eine solche Unterscheidung auch für visuelle Ansätze beim Entwickeln von Benutzungsschnittstellen wünschenswert und wird hier eingeführt:

Die nun folgenden vier Ebenen der Programmierung beziehen die angesprochenen Lernkurven und Abstraktionsmechanismen in Programmierumgebungen ein (die Mechanismen werden in Abschnitt 4 im einzelnen vorgestellt). Die Abstraktionsebenen von Programmierumgebungen, die visuelle Ansätze integrieren, werden in dieser Arbeit wie in Abbildung 2.5 klassifiziert.

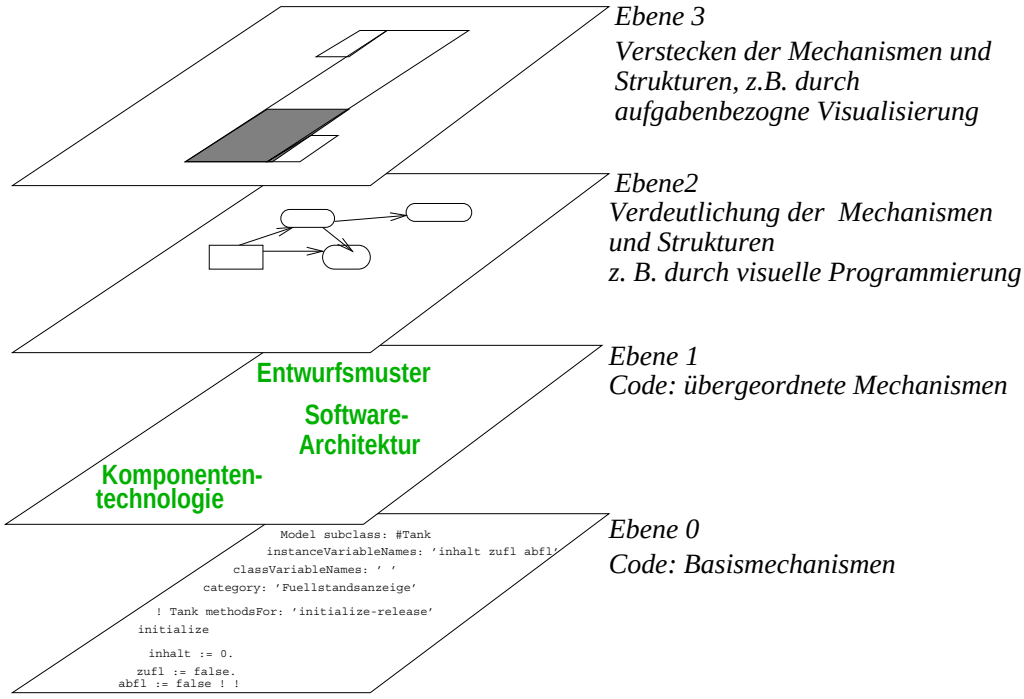


Abbildung 2.5: Ebenen der Programmierung

Aufgabenorientierte Programmierebene, Ebene 3: Die Programmiermechanismen und -strukturen werden durch an die Aufgabe angelehnte Mechanismen versteckt, z. B. in Anordnungswerkzeugen.

Strukturorientierte visuelle Programmierebene, Ebene 2: Programmiermechanismen und Strukturen werden visuell verdeutlicht, z. B. durch Datenfluß-Sprachen oder Entwurfsmuster-Diagramme.

Strukturorientierte textuelle Programmierebene, Ebene 1: Abstrakte textuelle Beschreibung oder Kapselung von Basismechanismen, durch Programmiermechanismen und -strukturen, z. B. Komponententechnologie und Entwurfsmuster. Details werden vor der Entwickler/in versteckt.

Basis-Programmierebene, Ebene 0: Ist die textuelle Programmierung von Klassen und Objekten ohne komplexe Mechanismen.

Durch die verschiedenen Programmierebenen entstehen Probleme beim Aufbau des konzeptuellen Modells der Entwickler/in. In Abschnitt 4.5 werden einige Programmierumgebungen vorgestellt, die visuelle Spezifikationsmethoden enthalten. Ein Problem ist, daß sie die verschiedenen Ebenen nicht klar trennen und den Entwickler/innen Lern- und Benutzungsschwierigkeiten bereiten. Die Gründe für diese Schwierigkeiten werden in Abschnitt 4.5 anhand einer Studie herausgearbeitet.

2.2.5 Vergleichskriterien für visuelle Methoden zur Entwicklung von Benutzungsschnittstellen: Der sog. kognitive Einordnungsrahmen von Green und Petre

Green und Petre haben die psychologischen Grundlagen des Programmierens und visueller Programmiersprachen analysiert [GP96] und verschiedene sog. kognitive Dimensionen identifiziert, mit denen die kognitiven Eigenschaften solcher visueller Sprachen beschrieben werden können. Dieser sog. „kognitive Einordnungsrahmen" [GP96] wird auch für zwei im Rahmen der vorliegenden Arbeit durchgeführten Studien verwendet: den Vergleich von graphbasierten und direktmanipulativen visuellen Sprachen (Untersuchung 1, siehe Abschnitt 2.2.6, [SW00]) und die Untersuchung über die Akzeptanz visueller Ansätze in den sog. visuellen Programmierumgebungen (Untersuchung 3, siehe Abschnitt 4.5.2 [Sie96]). Die von Green und Petre identifizierten Dimensionen sind:

- **Abstraktionsgrad:** Welches ist die kleinste und größte Abstraktionsmöglichkeit? Können Fragmente gekapselt werden?
- **Ähnlichkeit mit anderen Sprachen:** Welche grundlegenden Mechanismen, sog. Programmier-Spiele, müssen gelernt werden?
- **Konsistenz:** Wenn ein Teil gelernt ist, wieviel muß noch abgeleitet werden?
- **Fehlerbehaftung**: Welche syntaktischen Fehler schleichen sich immer wieder ein? (z. B. Vergessen von Semikolons, die falsche Anzahl von Klammern).
- **Mentale Anforderungen:** Hohe mentale Anforderungen erkennt man z. B. daran, daß sich die Entwickler/in nebenbei Notizen oder Diagramme auf Papier malt, um nicht den Überblick zu verlieren.
- **Versteckte Abhängigkeiten**: Beispiele für versteckte Abhängigkeiten sind Verbindungen ohne Pfeile, die bidirektional interpretiert werden, Abhängigkeiten zwischen Komponenten, die nicht explizit dargestellt werden, oder die nicht sichtbaren Beziehungen zwischen den Zellen einer Tabellenkalkulation.
- **Zwang zu verfrühten Entwurfsentscheidungen:** Muß die Entwickler/in Entscheidungen treffen, bevor sie genug darüber weiß? Das ist z. B. der Fall, wenn das Layout eines Klassendiagramms nicht automatisch erstellt wird, sondern die Entwickler/in von Anordnung von Anfang an bedenken muß, daß neue Klassen eingefügt werden müssen. Das ergibt oft den sog. „Spaghetticode".
- **Zwischenergebnisse:** Ist es möglich, den Entwurf während der Entwicklung auszuprobieren?
- **Änderungsfreundlichkeit:** Wieviel muß an der visuellen Darstellung insgesamt geändert werden, um eine Änderung an einer Stelle zu ermöglichen? Zieht sich eine Änderung zäh durch das ganze System?
- **Überblick:** Wie gehört das Ganze zum Einzelnen, ist zu jedem Zeitpunkt jeder Teil sichtbar?

- **Sekundäre Notation:** Als sekundäre Notation werden Attribute wie Farbe oder die Bedeutung der Anordnung bezeichnet.

Die Eigenschaften einer visuellen Sprache stellen dann einen Punkt in diesem 13-dimensionalen Raum dar und sind somit mit anderen Sprachen vergleichbar.

2.2.6 Visuelles Programmieren auf den verschiedenen Programmierebenen

Oft werden für aufgabenorientierte visuelle Programmierung (auf der aufgabenorientierten Ebene, Fall A) andere Notationen verwendet als für die visuelle Programmierung von Strukturen und Mechanismen (auf der Strukturebene, Fall B):

> **Fall A:** Die *visuelle aufgabenorientierte Programmierung* auf Ebene 3 ist *oft deklarativ*, d. h. durch Manipulation von graphischen Elementen wird angedeutet, **was** programmiert werden soll, aber nicht **wie** programmiert werden soll.
> Zu dieser Art von Programmierung gehört z. B. Programmieren durch Vormachen [Mye88, CBY00] und Programmieren durch Skizzieren [PAFJ00, LM00]. Diese Art von Programmierung ist dann besonders effektiv, wenn sie anwendungsspezifisch ist, d. h. wenn die Visualisierung Dinge aus dem Anwendungsgebiet zeigt, denen durch Manipulation Verhalten zugeordnet werden kann (in der Automatisierungstechnik z. B. Teile einer Anlage oder Blockdiagramme, siehe auch Kapitel 3).

> **Fall B:** Die *visuelle Programmierung der Struktur* von Programmen auf Ebene 2 benutzt eher *graphorientierte Visualisierungen*. Zu dieser Art von Programmierung gehört das Programmieren mit den bekannten Zustands-Übergangsdiagrammen oder Programmablaufplänen (auch bekannt unter dem Namen Flußdiagramme).

Der prinzipielle Unterschied zwischen beiden Programmierarten und deren Umsetzung soll an dem folgenden Beispiel verdeutlicht werden, das im Rahmen dieser Arbeit entwickelt wurde.

Untersuchung 1: Aufgabenorientiertes vs. strukturorientiertes Programmieren einer Robotersteuerung

In diesem Beispiel geht es zunächst nicht um die Entwicklung von Benutzungsschnittstellen, sondern um die Programmierung der Steuerung eines Roboters. Es wurden zwei Programmierarten untersucht:

- Programmieren durch auf der Aufgabenebene in der Umgebung LLDemo, die von der Autorin entworfen und von Yongmei Wu programmiert wurde [SW00] (Fall A: siehe auch Abschnitt 2.3.1) und
- Programmierung auf Strukturebene durch Flußdiagramme mit Hilfe der Programmiersprache LLWin [fis96] (Fall B).

These der Untersuchung

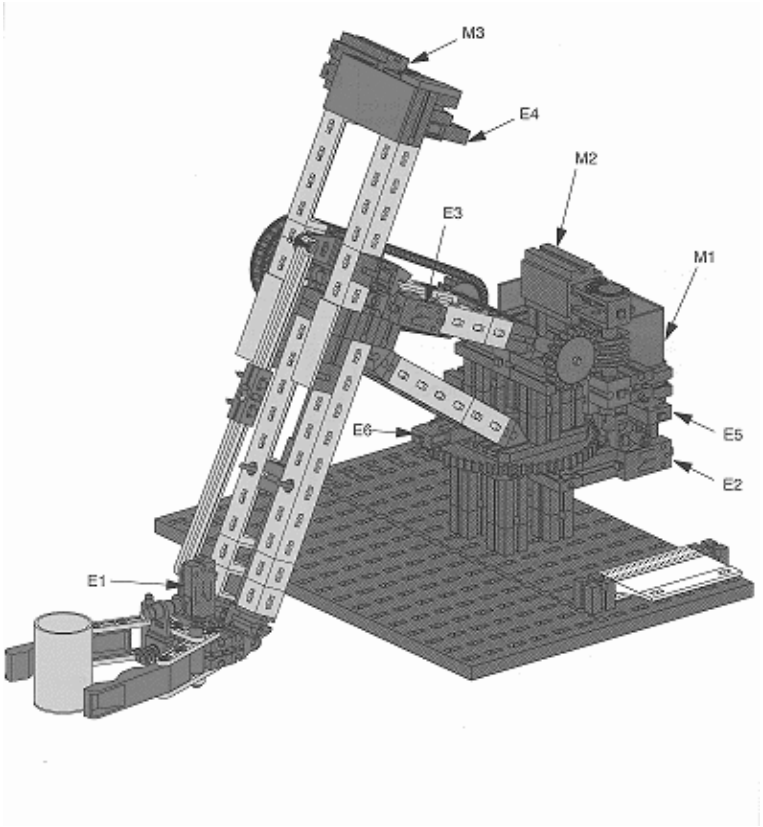
Aufgabenorientierte Programmierung ist leichter zu erlernen und benutzen. Daher wird vermutet, daß die aufgabenorientierte Programmierung in allen Fällen, in denen kein tieferes Verständnis erfolderlich ist, von den Testpersonen bevorzugt werden.

Da sie nicht zum Verständnis der grundlegenden Mechanismen beiträgt, sollten Testpersonen, die am Aufbau des Roboters interessiert sind, die **strukturorientierte Programmierung** bevorzugen.

Die Studie

Die Versuchsanordnung war wie folgt:

Der nebenstehend abgebildete Roboterarm wird über die Motoren M1, M2 und M3 angetrieben, die wiederum von den Schaltern E1 - E6 ein- und ausgeschaltet werden können. Der Arm kann sich um 225 Grad drehen und den Greifer 60 mm heben. Der Greifer kann zum Greifen und Loslassen geöffnet und geschlossen werden.



Als Beispiel soll folgende Bewegung programmiert werden:

> *Bewege den Arm um 45 Grad mit geöffnetem Greifer, senke den Greifer ab und greife ein Objekt, das ca. 1 cm breit ist, bewege den Arm um 45 Grad zurück und stelle das Objekt ab.*

LLWin stellt Blöcke zur Verfügung, mit denen die Motoren und Schalter einzeln gesteuert werden können. Das Programm für das Beispiel sieht programmiert so aus, wie in Abbildung 2.6 dargestellt.

Um das Beispiel zu programmieren, muß sich die Programmierer/in also genau mit den Verhaltensmechanismen des Systems, also des Roboterarms, auskennen (Ebene 1).

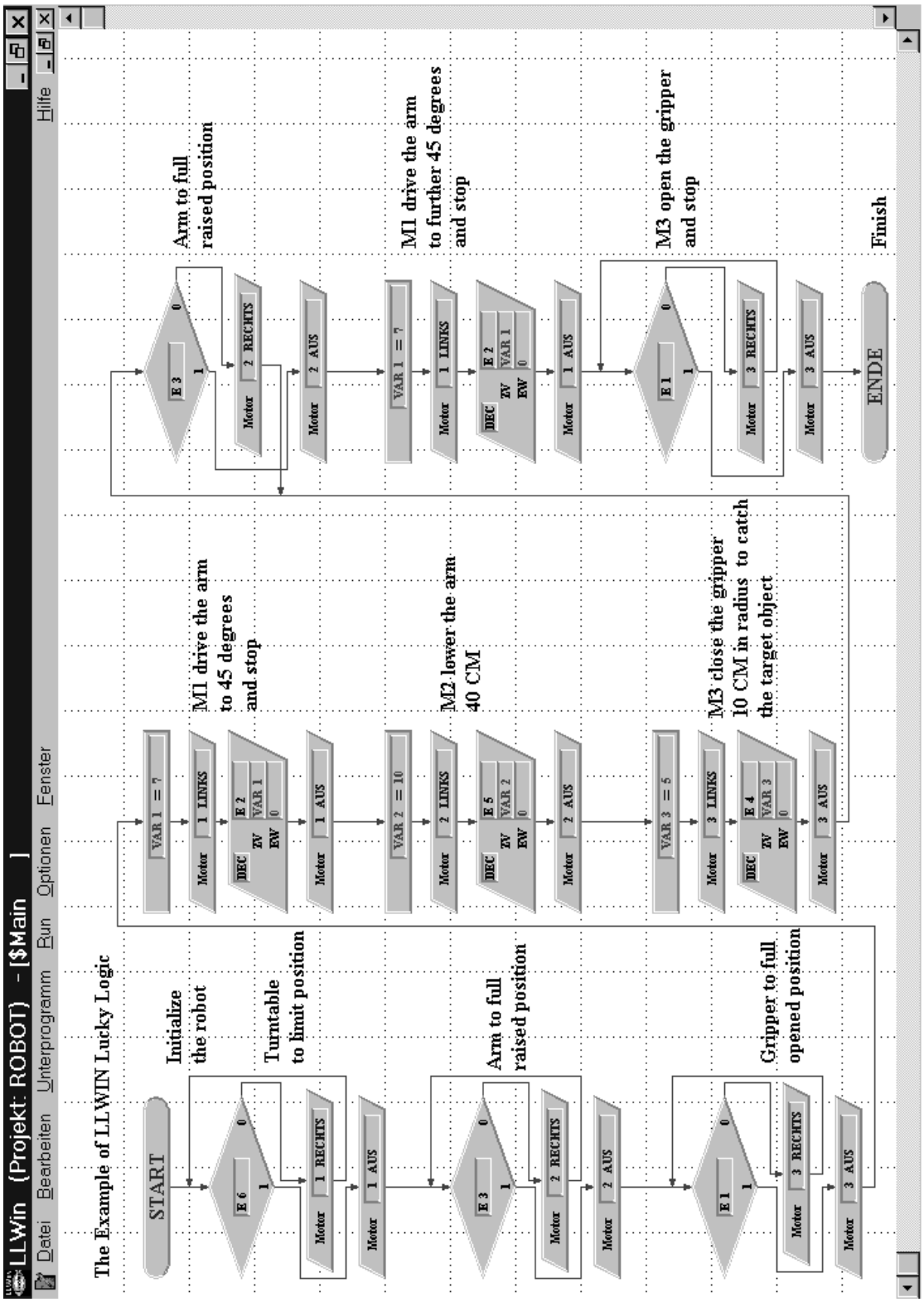


Abbildung 2.6: Roboterprogramm in LLWin

Bei der Programmierung in der Umgebung LLDemo wird dagegen nur die Bewegung spezifiziert und die Umsetzung in die verschiedenen Motorsteuerungen erfolgt automatisch (Ebe-

ne 2). Dabei wird die Fläche, die der Roboterarm mit dem Greifer erreichen kann, als Ausschnitt einer Kreisfläche dargestellt und der Greifer als Punkt. Die Höhe wird in ein Textfeld eingegeben, der Greifer über ein Menü geöffnet und geschlossen. Die Programmierung erfolgt durch Aufzeichnen einer Steuerungssequenz. Abbildung 2.7 zeigt Ausschnitte aus der Sequenz.

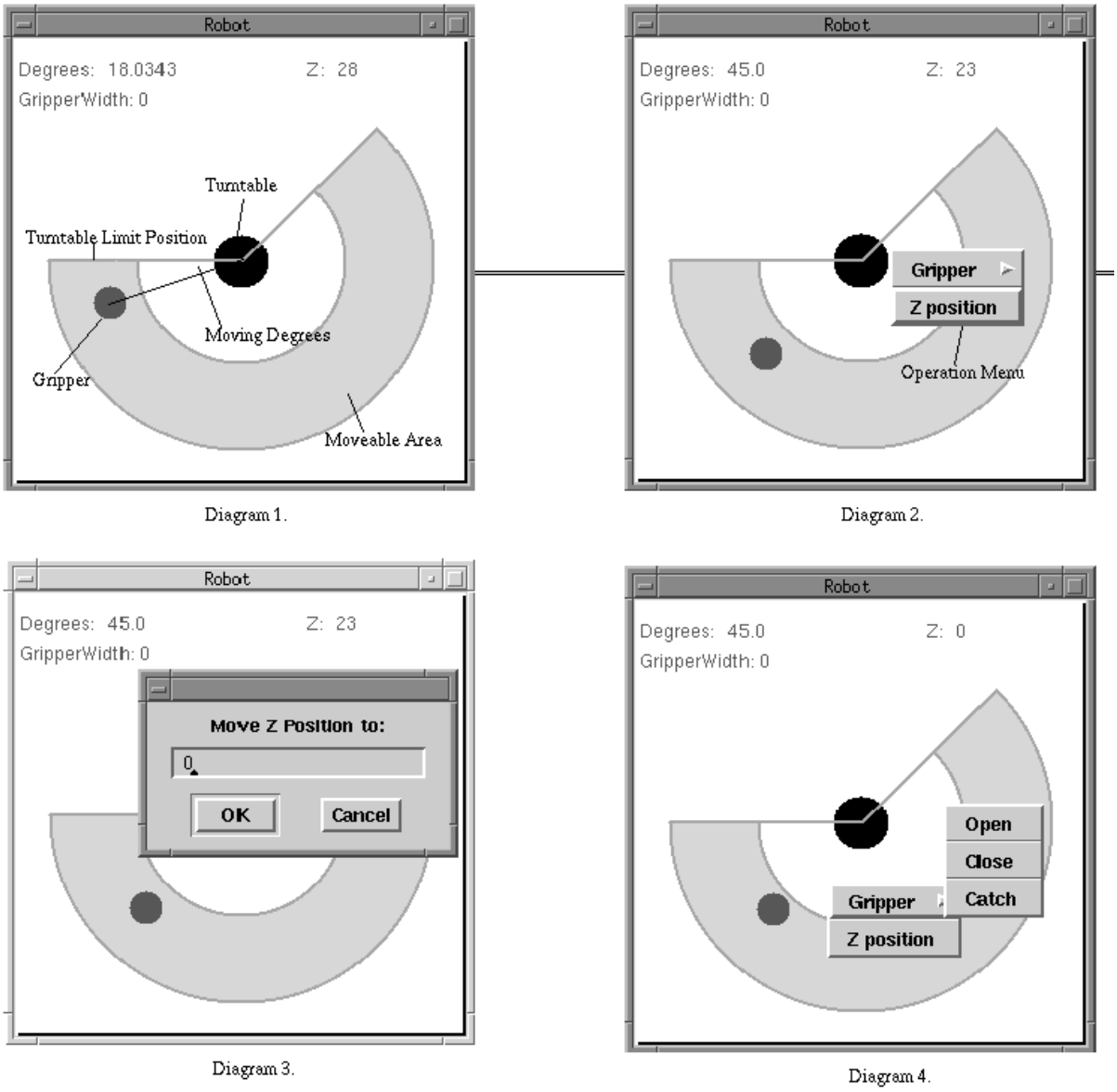


Abbildung 2.7: Steuerung des Roboters durch direkte Manipulation und Menüs auf Anwendungsdarstellungen in LLDemo

Abbildung 2.8: Die Versuchsanordnung

Bewertung der beiden Programmiermethoden

Vorteile

LLWin	**LLDemo**
Sehr genau. Jeder Schalter und Motor kann genau angesprochen werden und damit wird die Bewegung sehr exakt.	Einfach zu lernen und anzuwenden. Wenn die Benutzer/in die Benutzungsschnittstelle verstanden hat, kann sie den Roboter direkt durch Mausbewegungen steuern. Sie muß die Hardware nicht verstehen.
Die visuelle Sprache ist einfacher als eine textuelle Sprache.	Sehr interaktiv. Die Programmierung erfolgt bei Ausführung des Beispiels, man muß das Programm nicht im Voraus planen ohne die Bewegungen zu sehen.
Die Terminologie ist an die Hardware angepaßt. Deshalb ist die Programmierung für Hardware-interessierte Benutzer/innen gut verständlich.	Zur Laufzeit kann in das gespeicherte Programm eingegriffen und es kann verändert werden.
	Die Programmierung ist einfach, da nur nach Beendigung der Bewegungen die Bewegung als Programm abgespeichert werden muß.

Nachteile

LLWin	**LLDemo**
Jede Interaktion muß vorher geplant werden (ohne die Bewegung zu sehen).	Es ist schwer, eine Bewegung mit der Maus genau zu spezifizieren.
Die Benutzer/in muß den Aufbau des Roboters sehr genau kennen, um eine Bewegung aus der Steuerung der verschiedenen Motoren zusammensetzen zu können.	Beim Programmieren muß die Benutzer/in sehr konzentriert sein, um z. B. Kollisionen zu vermeiden.
Eine programmierte Bewegung kann nicht zur Laufzeit geändert werden.	Wenn etwas am Aufbau der Anlage geändert wird, muß die Übersetzung von Mausbewegung in Roboterbewegung umprogrammiert werden. Wenn die Benutzer/in den Aufbau nicht kennt, muß das von einer anderen Person gemacht werden, oder die Benutzer/in muß sich doch mit dem Aufbau der Anlage beschäftigen.
	Die Wahl zweidimensionaler Objekte als Repräsentation einer dreidimensionalen Realität macht die Interaktion schwierig.

Neben diesen offensichtlichen Vor- und Nachteilen sollte geprüft werden, wie die beiden Programmiermethoden von Benutzer/innen angenommen werden, insbesondere ob die Benutzer/innen lieber den Aufbau der Anlage kennen oder mit der bloßen Manipulation des Roboters zufrieden waren. Dazu wurde mit sieben Student/innen der Vorlesung „Entwurf interaktiver Programme" ein Experiment durchgeführt, das im folgenden beschrieben wird:

Variablen

In Anlehnung an den in Abschnitt 2.2.5 vorgestellten kognitiven Rahmen zur Bewertung visueller Methoden, wurden folgende zu untersuchende Variablen identifiziert:

- **Einfachheit der Programmierung**: Diese Variable läßt sich durch die Zeit messen, die die Benutzer/innen brauchen, um die Sprachen zu erlernen und ein Beispielprogramm durchzuführen.
- **Intuitivität**: Hierzu wurden den Benutzer/innen Fragen über die Eigenschaften der Sprachen, z. B. Visualisierungsgrad, benötigte mentale Fähigkeiten, Übersichtlichkeit und Benutzungsfreundlichkeit gestellt.
- **Exaktheit der erstellten Programme**: Die Exaktheit wurde daran gemessen, ob die Programme die Aufgaben erfüllten und der „Gradheit" der Wege, die der Greifer zurücklegte.
- **Leichtigkeit von Modifikationen**: Zur Bestimmung dieser Variablen wurde die Zeit zum Lösen der Aufgaben gemessen. Zusätzlich sollten die Benutzer/innen ihre persönliche Bewertung geben.
- **Konsistenz der Terminologie**: Um herauszufinden, ob die Benutzer/innen die Terminologie der beiden Ansätze verstanden haben, wurde die indirekte Frage verwendet, für welche Benutzer/innen das jeweilige System am besten geeignet ist. Außerdem wurde gefragt, welche textuelle Sprache dem jeweiligen System vermutlich zugrundeliegt.

Da die Hypothese war, daß die einfachere Benutzung von LLDemo durch die Sichtbarkeit der Anlagenstrukturen und -mechanismen bei LLWin aufgewogen wird, wurde erwartet, daß die beiden Werkzeuge gleich gute Bewertungen bekamen. Diese Hypothese wurde jedoch nicht bestätigt, was die folgenden Ergebnisse zeigen.
Zur Durchführung sollten die Student/innen nach jeweils einer Einarbeitungszeit das oben beschriebene Beispiel programmieren und nach der Beendigung eine Modifikation im Bewegungswinkel einfügen.

Ergebnisse der Studie

- **Einfachheit der Programmierung**: Die gemessenen Zeiten zeigen deutlich, daß LLDemo schneller zu erlernen und zu programmieren ist.

	Durchschnitt	bester Wert	schlechtester Wert
Einarbeitung Roboter	17 min	10 min	21 min
Einarbeitung LLWin	14 min	13 min	20 min
Aufgabe LLWin	59 min	51 min	83 min
Änderung LLWin	60 sec	30 sec	94 sec
Einarbeitung LLDemo	11,5 min	5 min	17 min
Aufgabe LLDemo	4 min	2 min	7 min
Änderung LLDemo	entfällt, da immer neue Aufgaben		

- **Intuitivität**:
  - Die Student/innen fanden LLDemo „visueller" und interaktiver als LLWin. Der Schritt, Flußdiagramme zu erstellen, wurde als zusätzliche Abstraktion verstanden. Die visuellen Objekte (siehe Abschnitt 2.3.1) von LLDemo wurden als der Anlage angemessen und natürlich zu benutzen empfunden.
  - LLWin benötigt mehr kognitive Fähigkeiten. Bei LLWin muß jeder Schritt explizit geplant werden, und die Flußdiagramme können leicht zu Spaghetticode werden.
  - Zusätzlich zum Verständnis der Anlage müssen bei LLWin Programmierinterna verstanden werden, z. B. die Abbildung von Variablen zu Motorzuständen.
  - Für Programme dieser Größe bieten beide Programmierarten einen guten Überblick und genügend klare Präsentation.
  - Subjektiv bevorzugten die meisten Student/innen (6 von 7) LLDemo weil „man keinen Code schreiben muß". Auf einer Skala von 0 bis 10 bekam LLDemo den Durchschnittswert 8.2 für Intuitivität, LLWin dagegen nur 5.1. Scheinbar ziehen die Student/innen die direkten Ergebnisse aus der Benutzung von LLDemo der genaueren Kenntnis der Anlage bei LLWin vor.

- **Exaktheit der erstellten Programme**: Erstaunlicherweise empfanden einige der Student/innen die mit LLDemo erstellten Programme exakter als die mit LLWin, obwohl in LLWin genaue Pfade angegeben werden können, und die Genauigkeit in LLDemo von der Mausführung abhängt.
  Das Problem ist hier, daß z. B. das Umsetzen von Winkeln in Impulsschritte eines Motors nicht sehr einfach zu berechnen ist und deshalb die LLWin-Programme oft fehlerhaft waren und mehrfach ausprobiert werden mußten.

- **Leichtigkeit von Änderungen und Modifikationen**: Kleinere Änderungen (z. B. Änderung eines Parameters oder einer Variablen) konnten in LLWin sehr schnell vorgenommen werden, obwohl das Programm jedesmal neu gestartet werden mußte. Änderungen in LLDemo hatten jedesmal eine Neuprogrammierung zur Folge, da eine Repräsentation der einzelnen Schritte nicht vorhanden war.

- **Konsistenz der Terminologie**: Hier wurde eine Differenzierung zwischen sichtbarer und transparenter Struktur erwartet, z. B. in dem Sinne, daß LLWin für Benutzer/innen empfohlen wird, die die Anlagenstruktur kennen, LLDemo dagegen für reine Anwender/innen. Diese Differenzierung erfolgte kaum (in nur einem Fall). Die Student/innen orientierten sich vielmehr an den benötigten kognitiven Anforderungen zum Erledigen der Aufgabe.

Bewertung der Ergebnisse

Nach Abschluß der Studie wurde deutlich, daß die Ergebnisse aussagekräftig sind, aber gleichzeitig zwei Größen bestimmen, nämlich

- erstens den Einfluß von direkter Manipulation auf Objekten des Anwendungsbereichs vs. einer diagrammartigen Programmierung und

- zweitens den Unterschied von Sichtbarkeit vs. Transparenz von Struktur und Mechanismen.

Vermutlich ist das der Grund für die Abweichung zwischen dem erwarteten und dem erreichten Ergebnis.

Es zeigt sich auch, daß eine die Vorteile beider Programmierarten ausnutzende Programmiersprache - am besten durch *Kombination der beiden Programmierarten* - eine gute Alternative wäre (z. B. LLDemo mit zusätzlicher schrittweiser Darstellung des Programms und Verfeinerung der einzelnen Schritte durch Ausprogrammierbarkeit der Umsetzung von Maus- in Roboterbewegung).

2.3 Prinzipien zur Unterstützung des Einsatzes visueller Spezifikationsmethoden als Grundlage für ein Werkzeug zur Entwicklung von Benutzungsschnittstellen

Aus der Untersuchung im vorhergehenden Abschnitt 2.2.6 und der noch folgenden Untersuchung über die Verwendung von visuellen Ansätzen zur Programmierung in Entwicklungsumgebungen (siehe Abschnitt 4.5.2) wird deutlich, daß nicht allgemeingültig beantwortet werden kann,

- ob visuelle Programmierung eingesetzt werden soll oder nicht,
- und ob es besser ist, dazu Methoden zur Verfügung zu stellen,
  - die die zugrundeliegende Struktur und Mechanismen verdeutlichen (Ebene 2)
  - oder solche, die eine aufgabenorientierte Manipulation auf Objekten der Anwendung realisieren (Ebene 3).

Welche Methoden sinnvoll sind, hängt sowohl von dem Wissensstand der Entwickler/in über die Struktur und Mechanismen ab, als auch von ihrer Veranlagung (visuell, auditorisch, haptisch, ...). Für ein Werkzeug folgt daraus, daß es der Entwickler/in die Wahl lassen soll, ob visuell oder textuell bzw. auf der aufgabenorientierten Ebene (Ebene 3), der Ebene der visuellen Darstellung von abstrakten Strukturen und Mechanismen (Ebene 2), der Ebene der textuellen Darstellung von abstrakten Strukturen und Mechanismen (Ebene 1) oder auf der Ebene der Basismechanismen (Ebene 0) programmiert wird.

Um die Programmierung auf allen Ebenen anbieten zu können, muß das Werkzeug einige Grundprinzipien für den Einsatz visueller und textueller Methoden unterstützen. Diese Prinzipien lassen sich aus dem bisher Gesagten ableiten.

Im Rahmen dieser Arbeit werden diese Prinzipien folgendermaßen genannt:

- Prinzip 1: „Unterstützung mehrerer Sichten",
- Prinzip 2: „Unterstützung direkter Manipulation", insbesondere die Ausprägungen

  - „Ziehen und Fallenlassen"
  - „Auswählen und Beschreiben"
  - „Anordnen"

- Prinzip 3: „Unterstützung der Darstellung von Struktur und Mechanismen" (auch „Bauplan" genannt)
- Prinzip 4: „Auswahl auf einer Palette" (auch „Unterstützung des Verwendens vordefinierter Elemente" genannt)
- „Unterstützung durch Assistenten"

2.3.1 Prinzipien für den Einsatz visueller Methoden

Prinzip 1: Unterstützung mehrerer Sichten

Es sollte möglich sein, verschiedene visuelle Spezifikationsmethoden zu *kombinieren*. Ein Werkzeug muß in der Lage sein, die verschiedenen Methoden als Sichten zu integrieren [GMH98]. Dies wird durch das in Kapitel 5 vorgestellte Modell zur Aufteilung der Benutzungsschnittstelle in mehrere Aspekte ermöglicht.

Auch in anderen Disziplinen, z. B. in der Architektur, werden verschiedene Sichten auf einen Entwurf benutzt. Do zeigt in einer Studie mit 62 Designer/innen [Do95], daß verschiedene Sichten für verschiedene Entwurfskonzepte benutzt werden. Mit Entwurfskonzepten sind dabei die verschiedenen Dimensionen des Gestaltungsraums bzw. die verschiedenen Aspekte gemeint, wie z. B. räumliche Anordnung oder Beleuchtung. Solche Ergebnisse können zwar nicht direkt in die Informatik übertragen werden, zeigen aber grundlegende Präferenzen von Designer/innen beim Entwurf von Systemen.

Abbildung 2.9: Sichten auf die verschiedenen Gestaltungsdimensionen beim Entwurf von Architektur (Abbildung aus [Do95])

Prinzip 2: Unterstützung direkter Manipulation

Ein direktmanipulatives System (nach [Shn83])

- stellt die Objekte der realen Welt visuell dar und

- erlaubt das Erfüllen von Aufgaben durch Manipulation dieser Darstellungen.

Direkte Manipulation kann durch das *Object Action Interaction*-Modell (OAI) ausgedrückt werden [Shn98], wobei jedem Objekt eine begrenzte Zahl von Aktionen zugeordnet wird. Wenn die Benutzer/in sich mit dem Anwendungsgebiet auskennt, kann sie ohne Programmierung Aufgaben durch Interaktionen mit (z. B.) der Maus wie „Klicken und Ziehen und Fallenlassen" (engl.: *drag and drop*) erledigen.

> *"... if a user knows how to perform a task on the computer, that should be sufficient to create a program to perform the task"* [Cyp93]

Ein Beispiel ist Fall A der Roboterprogrammierung in Abschnitt 2.2.6.

Bei der Übertragung dieses Prinzips auf die Entwicklung von Benutzungsschnittstellen stellt sich wiederum die Frage, welches die „Objekte der realen Welt" sind (wie schon in Abschnitt 2.1.1 beschrieben, handelt es sich eigentlich um Abstraktionen, gedachte Objekte) und wie die Aufgabe, eine Benutzungsschnittstelle zu entwerfen, durch Manipulation einer visuellen Repräsentation zu lösen ist. Im folgenden werden verschiedene Ausprägungen von direkter Manipulation betrachtet, die als Grundlage für visuelle Spezifikationsmethoden (siehe Kapitel 6) dienen.

a) Ziehen und Fallenlassen

Dies ist die klassische Interaktion direkter Manipulation, die inzwischen in allen Werkzeugen direkter Abbildungen (WYSIWYG) realisiert ist. Es realisiert das Ausschneiden oder Kopieren eines Elements aus einer Umgebung und Einfügen des Elements oder der Kopie in eine andere Umgebung.

b) Auswählen und Beschreiben

„Auswählen und Beschreiben" sind zwei ebenfalls grundlegende Operationen direkter Manipulation, und zusammen sind sie die grundlegenden Operationen für visuelle Programmierung.

Zeigen - die Vorarbeit für Auswählen - ist das Deuten auf ein Element und erzeugt damit eine „gerichtete visuelle Aufmerksamkeit" ([HH92]). *Auswählen* bedeutet, sich das Element, auf das gezeigt wurde, zu merken oder markieren, und kann sequentiell mehrere Elemente erfassen.

Das *Auswählen* wird oft ergänzt durch textuelle, sprachliche oder sonstige Anmerkungen, deren Interpretation eine *Bedeutung* für oder *Beschreibung* zu den gezeigten Elementen ist.

Fast jede visuelle Spezifikationsmethode beruht letztendlich auf dem Prinzip von „Auswählen und Beschreiben", wie im folgenden gezeigt wird. Dazu werden die verschiedenen Arten und Ergebnisse von „Auswählen und Beschreiben" klassifiziert; diese Klassifikation vereinfacht später die Beschreibung der Methoden.

Auswahlarten Beim Arbeiten mit Computern gibt es sprachliche bzw. textuelle, sowie visuelle zweidimensionale und dreidimensionale Auswahltechniken.

- **Sprachliches Auswählen** geschieht mittels Ausdrücken wie „Das rote Viereck" oder „das da" [Bol80] oder durch Positionsangaben wie z. B. „das Element auf 34 @ 45". Ähnliche Ausdrücke können auch textuell eingegeben werden.

- Zu den **räumlichen zweidimensionalen Auswahltechniken** gehört das Auswählen, z. B. durch Positionsänderung oder Klicken mit der Maus oder ähnlichen Eingabegeräten, die in der Lage sind, eine Position in einer Ebene anzugeben.

- Die dem Menschen natürliche Interaktion ist **dreidimensional** und wird durch Gesten ausgedrückt (siehe z. B. [JDM99]). Ein Computer muß entweder in der Lage sein, die Gesten zu verfolgen oder er muß Eingabegeräte wie einen berührungsempfindlichen Bildschirm haben.

Für die visuellen Methoden, die in dieser Arbeit vorgestellt werden, werden ausschließlich die Maus und die Tastatur benutzt, d. h. zweidimensionales Auswählen.

Beschreibungsarten Wie oben erwähnt, wird die Definition durch die Interpretation des Auswahlvorgangs im Zusammenhang mit darauf folgenden Aktionen festgelegt.

- **Beschreibung ohne Parameter:** die Interpretation kann nur aufgrund eines Modus (oder Zustands) erfolgen, in dem das Spezifikationswerkzeug sich befindet. Ein Beispiel ist ein Werkzeug, mit dem Graphen gezeichnet werden können: Zum Erstellen einer Verbindung zwischen zwei Knoten wird der Verbindungsmodus gewählt und dann die beiden Knoten ausgewählt. Dadurch wird die Interpretation „Verbindung herstellen" festgelegt. Die gleiche Interpretation kann auch nach der Selektion der beiden Knoten erfolgen, wobei man dann den Modus auch als Parameter verstehen könnte.

- **Beschreibung mit Parametern:** die Interpretation erfolgt mit Hilfe zusätzlicher Information, z. B. Anmerkungen [Lie93a][Lie93b], einfache Werte oder Aktionen (z. B. Skripte oder Programme bzw. Programmstücke). Solche Skripten und Aktionen können textuell oder visuell angegeben werden.

Ergebnisse von Auswählen und Beschreiben „Auswählen und Beschreiben" erfolgt immer auf einer visuellen Darstellung. Die Bedeutung oder Beschreibung, die durch „Auswählen und Beschreiben" zu einer visuellen Darstellung gefügt wird, kann Teil dieser Darstellung sein oder nicht:

- **Bedeutung oder Beschreibung wird Teil der visuellen Repräsentation:** Zu dieser Art von Spezifikationsmethoden gehören z. B. Graphenmethoden: Durch "Auswählen und Beschreiben" werden die Verbindungen zwischen den Knoten hergestellt, die z. B. als Methodenaufrufe oder Ereignisverteilung (siehe Abschnitt 4.3) interpretiert werden können.

- **Bedeutung oder Beschreibung ist Metainformation, ist aber in der gleichen visuellen Darstellung enthalten**: Ein Beispiel sind die in [Lie93a] vorgestellten Anmerkungen in Dokumenten, z. B. als Zusatzinformation zu Gebrauchsanweisungen. Ein Beispiel aus der Spezifikation von Programmen sind Markierungen, die beim Verwenden von Entwurfsmustern in Klassendiagramme eingefügt werden. Solche Markierungen zeigen an, welche Rolle eine Klasse in einem Entwurfsmuster spielt (siehe Abschnitt 6.6).

- **Bedeutung oder Beschreibung ist Metainformation, wird aber in die visuelle Darstellung nicht aufgenommen**: z. B. Eigenschaften, die die Verwaltung eines Diagramms betreffen, wie der Dateiname, unter dem das Diagramm abgespeichert wird (in einigen Werkzeugen allerdings auch sichtbar).

c) Anordnen

Die Anordnungs-Manipulation ist eigentlich ein Spezialfall von „Auswählen und Beschreiben". Graphische Benutzungsschnittstellen-Elemente werden auf einer Arbeitsfläche angeordnet, d. h. als Beschreibung wird automatisch die Posititionsangabe aus der Anordnung abgeleitet. Im Gegensatz zu „Ziehen und Fallenlassen" wird die Anordnung explizit festgelegt und nicht dem Kontext überlassen.

Prinzip 3: Unterstützung der Darstellung von Mechanismen und Strukturen - der Bauplan

Wie in Abschnitt 2.2.6 dargestellt, ist es für den Entwurf von Benutzungsschnittstellen wichtig, ihre Mechanismen und Strukturen zu kennen und anwenden zu können. Diese Mechanismen und Strukturen müssen daher visualisiert werden; sie werden in Abschnitt 4 dargestellt.

Prinzip 4: Auswahl auf einer Palette

Paletten sind Menüs, deren Elemente Piktogramme sind. Diese Auswahlmethode kann durchgängig für viele visuelle Methoden in Kapitel 6 verwendet werden.

Prinzip 5: Unterstützung des Verwendens von Assistenten

Assistenten oder engl. *wizards* sind Sequenzen von Formularen oder Texten, die einer Benutzer/in helfen, Eingaben zu machen. Erstens wird durch die Sequenz genau festgelegt, wann welche Eingaben zu machen sind, und zweitens werden aus den Eingaben Code generiert. Viele Tätigkeiten beim Software-Entwurf allgemein und auch bei der Entwicklung von Benutzungsschnittstellen können durch solche Assistenten automatisiert werden. Meistens sind diese Assistenten jedoch eingepaßt in die jeweilige Entwicklungsumgebung (z. B. der *CodingAssistant* von VisualWorks) und abhängig von Klassenbibliothek und Programmiersprache (und deren Konventionen). Deshalb werden Assistenten in dieser Arbeit nur beispielhaft betrachtet, z. B. der *Design Pattern Assistant* zur Unterstützung der Anwendung von Entwurfsmustern, siehe 6.6.

Assistenten haben den Vorteil, daß sie die benötigte Information von der Benutzer/in erfragen, die Benutzer/in sich aber nicht darum kümmern muß, an welche Stelle die Information geschrieben wird oder welche Befehle nötig sind, um die Information einzutragen.

Kapitel 3

Benutzungsschnittstellen für Prozeßleitsysteme

Wie in der Einleitung erwähnt, waren die besonderen Anforderungen von Benutzungsschnittstellen für technische Systeme Ausgangspunkt dieser Arbeit. Der Entwurf von Benutzungsschnittstellen beispielsweise in Büroanwendungen oder für Datenbankabfragen ist bereits gut untersucht, diese Benutzungsschnittstellen können oft sogar aus den Datenmodellen generiert werden [Bal93]. Eine solche Unterstützung ist bei vielen technischen Anwendungen noch nicht gegeben. Dieses Kapitel soll am Beispiel Prozeßleitsysteme aufzeigen,

- wie sich technische Systeme von anderen Anwendungssystemen unterscheiden,
- wie sich die Anforderungen an die Benutzungsschnittstellen unterscheiden und
- wie sich die Anforderungen an den Entwurf von Benutzungsschnittstellen unterscheiden.

3.1 Der Unterschied zwischen technischen Systemen und anderen Anwendungssystemen

Besonderheiten der Überwachung von Automatisierungssystemen

Beispielhaft für das große Feld der technischen Systeme werden hier Benutzungsschnittstellen für Automatisierungssysteme im weiteren Sinne untersucht,d. h. Systeme, die einen Prozeß in einer Maschine oder Anlage steuern oder regeln. Abbildung 3.1 zeigt die Grundstruktur eines Automatisierungssystems (nach [Kie95]). Obwohl die Benutzungschnittstelle zum Leitsystem gehört, ist gerade der Prozeß, den das System steuert, von Bedeutung für das Aussehen der Benutzungsschnittstelle.

Unter dem Begriff *Automation* werden eigentlich „alle Fertigungsprozesse, die ganz oder teilweise unter maschineller Steuerung ohne permanente Eingriffe des Menschen ablaufen" [Sch97b] verstanden. *Automatisierung* wird dann definiert als „realtechnischer Vorgang, der zur Automation führt. Dabei werden Arbeits- und Produktionsprozesse so gestaltet, daß der Mensch weder permanent noch zu genau festlegbaren Zeitpunkten in den Funktionsablauf technischer Systeme einzugreifen braucht." Der Mensch ist bei solchen Systemen nicht mehr

Teil des Prozeßablaufs, sondern nur noch *Beobachter/in* oder *Bediener/in* der Maschine. Im Folgenden wird daher die Benutzer/in einer solchen Benutzungsschnittstelle Bediener/in genannt.

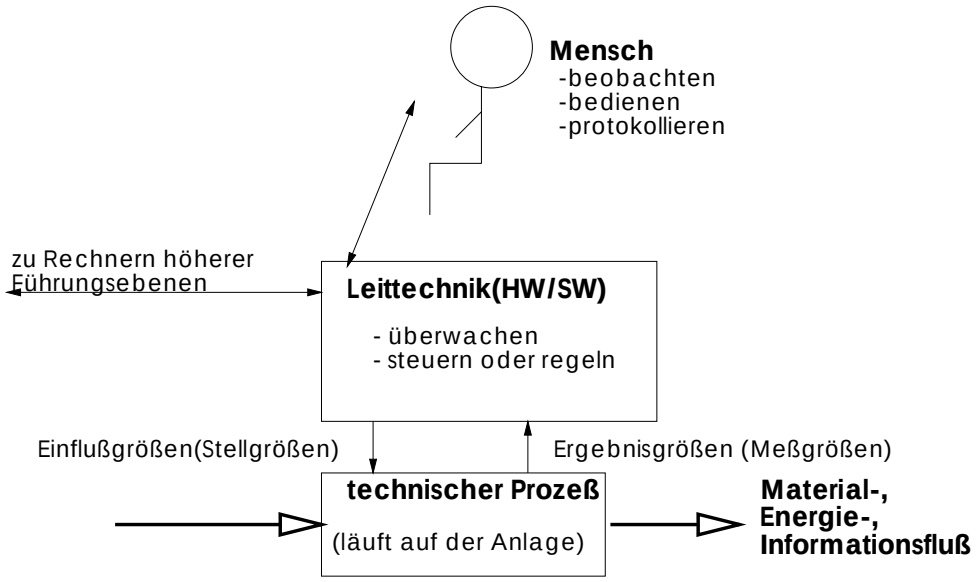


Abbildung 3.1: Grundstruktur eines Automatisierungssystems

Das ist eine wesentliche Unterscheidung zu anderen Informationssystemen, z. B. im Bürobereich: Beim Schreiben eines Dokuments beispielsweise ist die Schreiber/in in den Prozeß eingebunden, indem sie das Produkt erstellt. Das Drucken des Dokuments oder das automatische Generieren eines Dokuments dagegen sind automatisiert. Solche Automatisierungssysteme sind hier nicht gemeint, sondern Systeme, mit denen ein technischer Prozeß überwacht wird.

Der Entwurf solcher Systeme stellt besondere Anforderungen an die Entwicklung ihrer Benutzungsschnittstellen und den Softwareentwurf allgemein [Kie95]: Die Modellbildung für ein Automatisierungssystem bildet eine bestehende Anlage nach, im Gegensatz zu Informationssystemen, bei denen Abläufe komplett „neu erfunden" werden.

3.2 Der Unterschied zwischen den *Anforderungen an Benutzungsschnittstellen* in technischen Systemen und anderen Anwendungssystemen

Die darzustellende Information

Da es die Aufgabe der Bediener/in ist, den Prozeß der Maschine/Anlage zu beobachten und evtl. zur Bedienung einzugreifen, muß die Benutzungsoberfläche im wesentlichen den Status des Prozesses wiedergeben und ihr die Entscheidung ermöglichen, ob sie in den Prozeß eingreifen muß oder ihn sich selbst überlassen kann. Nach der Richtlinie 3850 des VDI/VDE zeigt die

Oberfläche daher entweder reaktionspflichtige, bestätigungspflichtige oder nichtbestätigungspflichtige Information. Diese wird in der Richtlinie in Informationsklassen eingeteilt (nach [Eis00]), hier mit abnehmender Priorität aufgelistet:

Normalbetrieb

- Statusinformation
- Hinweisinformation
- Bedienfehlerinformation
- Allgemeine Informationen
- Aufmerksamkeit erhalten

Störfall

- Störungsinformation
  - Alarm
  - Warnung

Die Priorisierung hat auch Einfluß auf die Positionierung der Information und führt zu typischen Anordnungen der Benutzungsschnittstellen (siehe 3.3.5 und 4.4.2).

Komplexität der Benutzungsschnittstellen

Obwohl die Hauptaufgabe der Benutzer/in das Beobachten ist, muß die Bediener/in auch in den Prozeß eingreifen können. Darüberhinaus muß die Maschine auch parametriert, konfiguriert und programmiert werden (z. B. NC-Maschinen). D. h., daß viele Funktionen der Maschine/Anlage von der Benutzungsschnittstelle aus aufgerufen werden können und der Prozeß damit auch durch die Benutzungsschnittstelle explizit vom Menschen gesteuert werden kann.

Dadurch besitzen die Benutzungsschnittstellen ebenfalls eine große Funktionalität, die strukturiert angeboten werden muß, damit die Bediener/in nicht überfordert wird.

Anforderungen an die Bediener/in und Implikationen für die Benutzungsschnittstelle

Die Schwierigkeiten für die Bediener/in im laufenden Betrieb ergeben sich daraus, daß es den „Normalbetrieb" und den „Störfall" gibt. Im Normalbetrieb muß die Information so übersichtlich sein, daß alle wesentlichen Informationen auf einen Blick gesehen werden können. Deshalb muß die Information geeignet zusammengefaßt werden, z. B. erlaubt die Analoganzeige eines Werts als Drehskala eine viel schnellere Klassifikation als die digitale Anzeige, wie beispielsweise an Uhren, Geschwindigkeitsanzeigern oder Temperaturdarstellungen erkennbar wird. Im folgenden Beispiel wird eine Zahl erstens durch sechs Informationseinheiten ausgedrückt, nämlich die sechs Ziffern, und zweitens durch eine Informationseinheit, nämlich den Zeigerstand, siehe Abbildung 3.2.

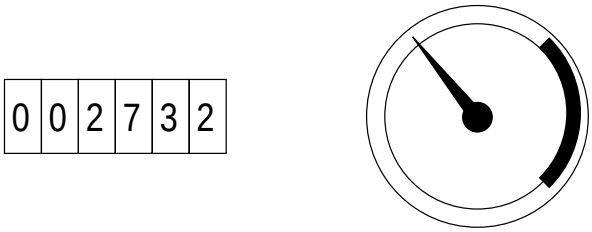


Abbildung 3.2: Analog- und Ziffernanzeige eines Werts der Anwendung

Wenn es nicht auf die Genauigkeit ankommt, sondern darauf, in einem Blick den ungefähren Stand zu erfassen, ist die zweite Darstellung zu bevorzugen. Da die Wahrnehmungsfähigkeit durch das Kurzzeitgedächtnis des Menschen auf die Verarbeitung von 7 ± 2 Einheiten begrenzt ist, kann auf diese Weise deutlich mehr Information dargestellt werden (siehe z. B. [Rös00]). Die Implikation ist, daß es möglich sein muß, Benutzungsoberflächenelemente zu verwenden, die solch eine Informationsverdichtung erlauben.

Außerdem muß die Benutzungsoberfläche so gestaltet werden, daß die Aufmerksamkeit der Bediener/in erhalten bleibt, d. h. sie darf nicht langweilig gestaltet werden.

Im Störfall muß die Bediener/in augenblicklich von dem Problem in Kenntnis gesetzt werden, d. h. die Oberfläche muß sich auffällig verändern. Dies kann visuell durch die Verwendung von Farbe, Farbumschlägen, Animationen (z. B. Blinken) oder durch akustische Signale geschehen. Daher müssen die Benutzungsoberflächen-Elemente mit solchem Verhalten belegt werden können.

Konzeptuelles Modell der Bediener/in eines Automatisierungssystems

Da der zu überwachende Prozeß für die Bediener/in das zentrale Element eines Automatisierungssystems ist, muß die Benutzungsschnittstelle diesen Prozeß in geeigneter Weise widerspiegeln. In größerem Maße als bei anderen Informationssystemen soll die Benutzungsschnittstelle transparent sein, d. h. für die Bediener/in soll der Prozeß wahrnehmbar sein und nicht die Benutzungsschnittstelle für sich. Das konzeptuelle Modell der Bediener/in wird sich zur Überwachungszeit daher ebenfalls am Prozeß anlehnen (das ist beim Programmieren der Maschine unter Umständen etwas anders). Das bedeutet, daß das konzeptuelle Modell der Bediener/in über den Prozeß bekannt sein muß. Es gibt viele Prozeßmodelle, die ganz bildhafte, einfache Modelle einschließen, wie z. B. die Vorstellung der physikalischen Abläufe in einer Maschine bei der Ausführung des Prozesses. Andere Prozeßmodelle schließen auch die mathematische und physikalische Modellierung des Prozesses oder das einfache Visualisieren der Ergebnisse des Prozesses ein. Wie eine Untersuchung ergab, die in Abschnitt 3.3.5 vorgestellt wird, hängt die gewünschte Visualisierung von diesem Modell ab. In Abschnitt 3.3.1 werden die verschiedenen Darstellungsformen diskutiert.

Realzeitanforderungen

Nach DIN 44300 bedeutet Realzeitverarbeitung: *„Ein Betrieb eines Rechensystems, bei dem Programme zur Verarbeitung anfallender Daten ständig betriebsbereit sind, derart, daß die Verarbeitungsergebnisse innerhalb der vorgegebenen Zeitspanne verfügbar sind. Die Daten können je nach Anwendungsfall nach einer zeitlich zufälligen Verteilung oder zu vorherbestimmten Zeitpunkten anfallen.*"

Eine Benutzungsschnittstelle, die ein solches Realzeitsystem bedient, muß daher die anfallenden Daten in Echtzeit auf dem Bildschirm anzeigen. Ebenso muß die Interaktion der Benutzer/in direkte Auswirkungen auf das System haben. Dazu gehört sowohl die Bearbeitung von zeitlich vorhersagbaren (synchronen) als auch von zeitlich nicht vorhersagbaren (asynchronen) Ereignissen.

Aufgrund der Rahmenbedingungen dieser Arbeit wurde der Realzeitaspekt nicht untersucht. Eine Einführung in die Modellierung von Echtzeitsystemen findet sich z. B. in [SGW94].

Zusammenfassung der Anforderungen an Benutzungsschnittstellen technischer Systeme

- Die Benutzungsschnittstelle soll den Prozeß, d. h. die Abläufe der Anlage/Maschine, widerspiegeln, d. h. die graphischen Elemente müssen in Form, Farbe und Verhalten flexibel an den Prozeß angelehnt werden können.
- Die Benutzungsberflächen von Automatisierungssystemen sind von der Anordnung der Elemente her verhältnismäßig statisch, von der darstellenden Graphik her dynamisch.
- Die hier betrachteten Benutzungsschnittstellen dienen zur Laufzeit zur Beobachtung/Bedienung von Anlagen, bzw. bei der Einrichtung zur Konfiguration der Anlagen. Daher bestehen sie meistens aus mehreren, vom Aufbau her jeweils ähnlichen, Bildschirmen. Im Gegensatz zu einem Textverarbeitungsprogramm oder einer Tabellenkalkulation kommt es nicht darauf an, daß die Benutzer/in den Inhalt konstruiert (wie einen Text oder ein Arbeitsblatt). Statt dessen sollen die Änderungen des Inhalts zur Beobachtung dargestellt bzw. der Inhalt bei der Bedienung angepaßt werden. Auch ist die Möglichkeit eher eingeschränkt, Schnittstellenelemente, die nicht direkt der Darstellung eines oder mehrerer Werte dienen, direkt zu manipulieren.

3.3 Der Unterschied zwischen den *Anforderungen an den Entwurf* der Benutzungsschnittstellen in technischen und anderen Anwendungssystemen

3.3.1 Darstellung von Anlagen durch die Benutzungsoberfläche in technischen Systemen

Anlagenmodelle sind entweder Prozeßmodelle, d. h. Beschreibung oder Nachbildung eines technischen Prozesses [Sch97b] - also der Beschreibung dessen, was in der Anlage passiert - oder Ausrüstungsmodelle, d. h. die Beschreibung der Zusammensetzung der Anlage.

Prozeßmodelle werden beschrieben durch

- Prozeßflußmodelle
- Prozeßzustandsmodelle
- Mathematische Modelle

- Ausnahmezustände
- Zugehörigkeit zu bestimmten Anlagenteilen

Ausrüstungsmodelle werden beschrieben durch

- Strukturmodelle, d. h. die Zusammensetzung der Anlage
- Zustandsmodelle der einzelnen Anlagenkomponenten
- Ausnahmezustände der einzelnen Anlagenkomponenten

Die Anlagen sollen durch die Benutzungsschnittstelle visualisiert werden. Neben den Anlagenmodellen fließen in den Entwurf der Visualisierung die Aufgaben ein, die die Benutzer/in der Prozeßvisualisierung hat. Dabei können die Oberflächen einerseits beliebig graphisch gestaltet werden, andererseits gibt es für die Darstellung von Prozessen auch standardisierte Bildelemente.

Freier Entwurf des Prozeßbildes als bildhafte Darstellung

Zwei Beispiele:

- Die wissenschaftliche Mitarbeiter/in in einem Labor einer Universität, die an der Regelung des Prozesses selbst - also an bestimmten Kenngrößen des Prozesses - interessiert ist, arbeitet mit einer Benutzungsschnittstelle, die den Status des Prozesses durch diese Kenngrößen anzeigt, z. B. als Funktionsgraph oder durch Zahlenwerte.

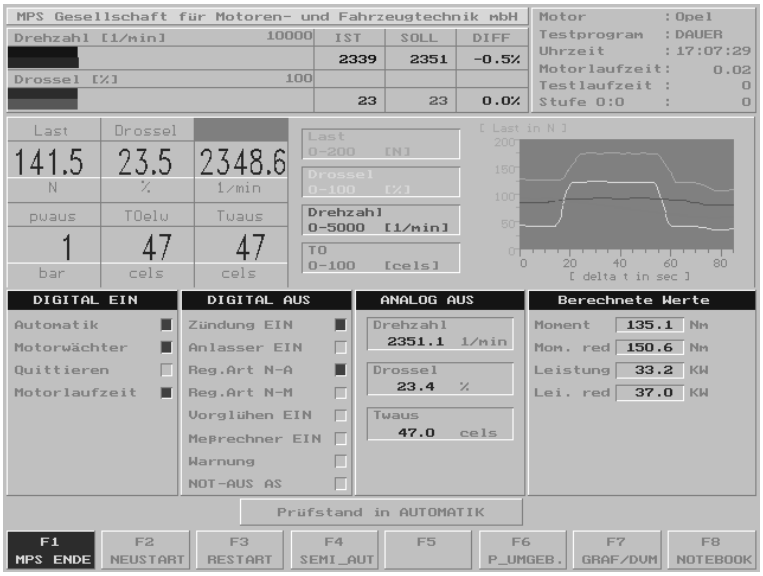


Abbildung 3.3: Prozeßvisualisierung eines Dieselmotors

- Die Überwacher/in der Automatisierung einer Molkerei wird dagegen die einzelnen Verarbeitungsstationen als Gerätebilder auf der Benutzungsoberfläche sehen, um im

Störungsfall den Fehler lokalisieren zu können. Benutzungsschnittstellen von modernen Kopierern fallen ebenfalls in die zweite Klasse.

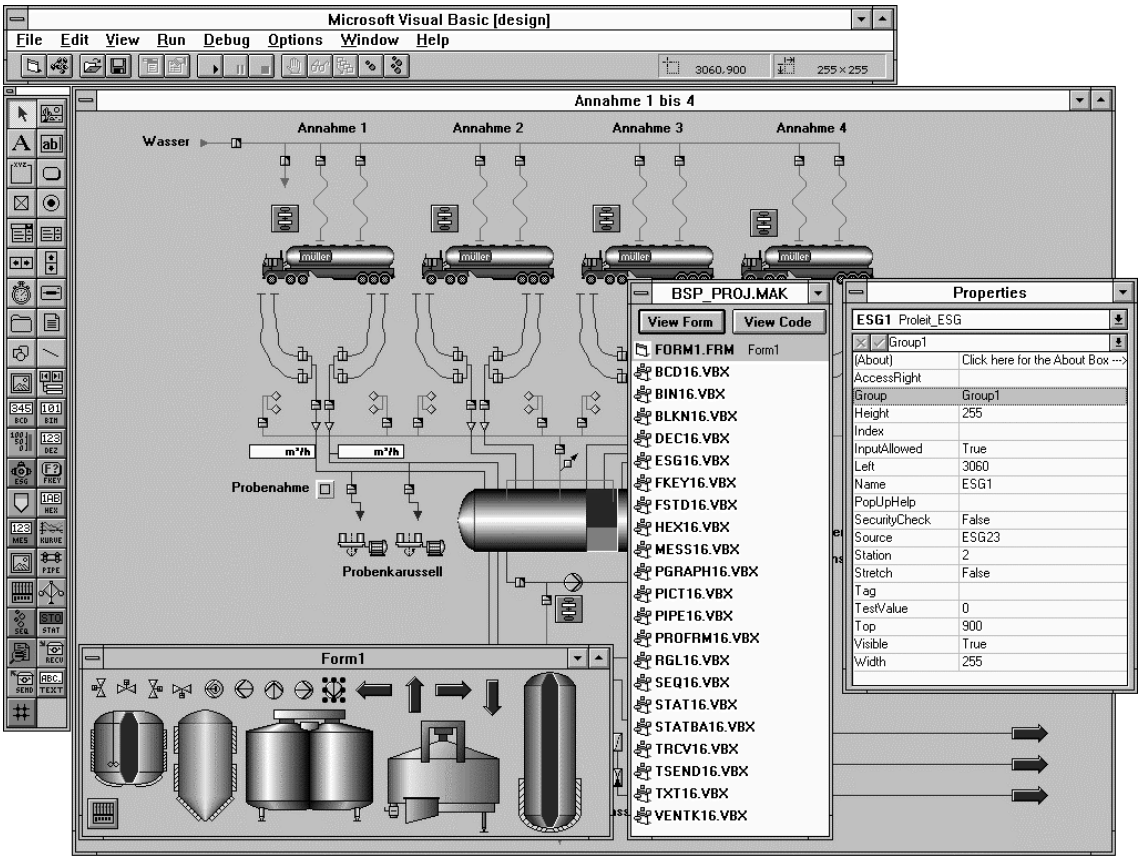


Abbildung 3.4: Prozeßvisualisierung der Steuerung der Milchwirtschaft in einer Region

Generell können Prozeßmodelle nach den Kriterien *Gewinnung des Prozesses*, *Zeitverhalten*, *Darstellung* und *Entwicklung* klassifiziert werden. Tabelle 3.1 zeigt mögliche Visualisierungsformen für die Prozeßmodelle.

Aspekt	Prozeßmodell	Visualisierung
Gewinnung	empirisches	Anzeige von Ein- und Ausgabewerten
	analytisches	Anzeige der Zustandsgrößen
Zeitverhalten	stationäres	Darstellung der Beharrungszustände
	dynamisches	Darstellung der Zustandsübergänge
Darstellung	mathematisches	Funktionsdarstellung
	gegenständliches	Darstellung der Anlage/Maschine
Entwicklung	lernendes	Darstellung des Lernzustandes
	adaptives	veränderliche Darstellung

Tabelle 3.1: Visualisierungsarten von Prozeßmodellen

3.3.2 Werkzeuge für den Entwurf von Benutzungsschnittstellen in technischen Systemen

Benutzungsschnittstellen für Informations- und Automatisierungssysteme werden heute in der Regel mit Hilfe von Fenstersystemen programmiert. Diese Programmierung erfolgt auf einem relativ niedrigen Abstraktionsniveau und erfordert Programmierkenntnisse von der Designer/in der Benutzungsoberfläche, sowie eine genaue Kenntnis der Schnittstellen zum Anwendungsprogramm. Werkzeuge wie User Interface Builder und Umgebungen wie User Interface Management Systeme erleichtern diese Arbeit, haben aber oftmals die folgenden Nachteile:

- Sie können nur einen festen Satz von Schnittstellenelementen erzeugen, z. B. Fenster, Knöpfe etc. Besondere Elemente, wie z. B. die Darstellung eines Roboters und das Verhalten dieser Darstellung müssen programmiert werden.
- Sie sind für spezielle Anwendungsbereiche gedacht und daher nicht universell verwendbar.
- Sie sind nicht intuitiv.
- Das Verhalten der graphischen Benutzungsschnittstellen-Elemente muß ausprogrammiert werden.

3.3.3 Besonderheiten der Zusammensetzung von Entwicklungsteams beim Entwurf von Benutzungsschnittstellen technischer Systeme

In technischen Anwendungen gibt es mehrere Personenkreise, die die Benutzungsschnittstellen entwerfen könnten, und Softwareentwicklungssysteme, die auf diese Personenkreise zugeschnitten sind:

- **Ingenieur/innen**
  bzw. diejenigen, die die Anwendung programmieren. Dies ist oft der Fall, wenn das System individuell für eine Anlage entwickelt wird. Dann entwickeln die Personen, die die Anlage konzipieren, auch die Software inklusive Benutzungsschnittstelle.
  Für diesen Personenkreis gibt es anwendungsspezifische Entwicklungssysteme sowie Standardsoftware. Ein Beispiel für ein Entwicklungssystem ist „FuelsManager", mit dem Tankstandsysteme aufgebaut und gesteuert werden können. Mit „FuelsManager" kann sowohl die Anlagesteuerung programmiert werden als auch die Benutzungsschnittstelle. Die Argumentation für solche Systeme liegt auf der Hand: *„Speziell entwickelte Tankstand-Software weist gegenüber Standard-Visualisierungspaketen mehrere Vorteile auf: Spezielle Volumenberechnungen, Transferoperationen und Tankgruppen sind Bestandteil des Produkts und brauchen nicht speziell für eine Anwendung entwickelt zu werden. Sind dann noch Grafikeditoren und Reportgeneratoren enthalten, ist das eine gute Kombination von Visualisierungspaket und speziellen Anforderungen für den Tankstand*" ([Käs99]). Der Nachteil solcher Entwicklungsumgebungen ist, daß sie nur für einen Anlagentyp verwendet werden können. Andere Entwicklungsumgebungen,

wie LabViews ([Nat00]), die ebenfalls anwendungsspezifisch konzipiert wurden, wurden später erweitert, um auch andere Anwendungsgebiete abzudecken. Insbesondere die Werkzeuge zur Entwicklung von Benutzungsschnittstellen wurden oft in anderen Anwendungsgebieten anzuwenden versucht, besonders dann, wenn das Ergebnis Quellcode einigermaßen verbreiteter Programmiersprachen wie C, C++ oder Java war (siehe dazu auch die Interviews in Abschnitt 3.3.5).
Bei der Verwendung von Standardsoftware wird das fertige Leit- oder Kontrollsystem für die Anlage als Software-Paket gekauft und nur noch entsprechend der speziellen Parameter der Anlage konfiguriert. Dann gibt es entweder
⋆ eine bereits festgelegte Benutzungsschnittstelle oder
⋆ die Benutzungsschnittstelle kann vom Konfigurationsteam zusammen mit den zukünftigen Benutzer/innen der Anlage erstellt werden.
In beiden Fällen (anwendungsspezifische Entwicklungsumgebung und Standardsoftware) sollte bei der Integration von Benutzungsschnittstellen-Werkzeugen mit HCI-Expert/innen und Informatiker/innen zusammengearbeitet werden. Oft sind moderne Methoden der Benutzungsschnittstellen-Entwicklung nicht enthalten, weil die Werkzeuge nach dem gleichen Schema entwickelt werden, wie der Anwendungsteil, d. h. ohne Beteiligung und Wissen der HCI-Expert/innen und Informatiker/innen.

- **HCI-Expert/innen**
  HCI-Expert/innen kennen den grundsätzlichen Aufbau ergonomischer Benutzungsschnittstellen, sind aber oftmals weder mit dem Anwendungsgebiet noch mit der Programmierung von Benutzungsschnittstellen vertraut.

- **Informatiker/innen** Informatiker/innen kennen sich mit Methoden des Software Engineering und der Programmierung allgemein gut aus. Das Anwendungsgebiet kennen sie nicht im Detail und weitergehende Kenntnisse über Software-Ergonomie haben sie in der Regel nicht.

Je nach Art des zu entwickelnden Systems können sich Entwicklungsgruppen dann mit Kombinationen aus Ingenieur/innen, HCI-Expert/innen und Informatiker/innen zusammensetzen:

- Das System wird individuell für eine Anlage entwickelt und die Aufteilung der Entwicklungsgruppen erfolgt z. B. nach Funktionsbereichen der Anlage, d. h. dieselben Personen, die den Aufbau der Anlage konzipieren, entwickeln auch die Software inklusive Benutzungsschnittstelle.

  – Das Leit- oder Kontrollsystem für die Anlage wird als Software-Paket gekauft und nur noch entsprechend der speziellen Parameter der Anlage konfiguriert. Dann gibt es entweder

    * eine bereits festgelegte Benutzungsschnittstelle oder
    * die Benutzungsschnittstelle kann vom Konfigurationsteam zusammen mit den zukünftigen Benutzer/innen der Anlage erstellt werden.

3.3.4 Spezifische Entwurfsmethoden der Entwicklung technischer Systeme und ihr Einsatz bei der Entwicklung von Benutzungsschnittstellen technischer Systeme

Die in technischen Anwendungsgebieten benutzten Methoden zum Entwurf und zur Beschreibung von Systemen entsprechen den in Abschnitt 2.2.6 abgeleiteten visuellen Darstellungsprinzipien.

Standardisierte Darstellungen

In technischen Anwendungsgebieten werden für bestimmtes Verhalten oft Standardsymbole verwendet, wie in Abbildung 3.5 zu sehen ist.

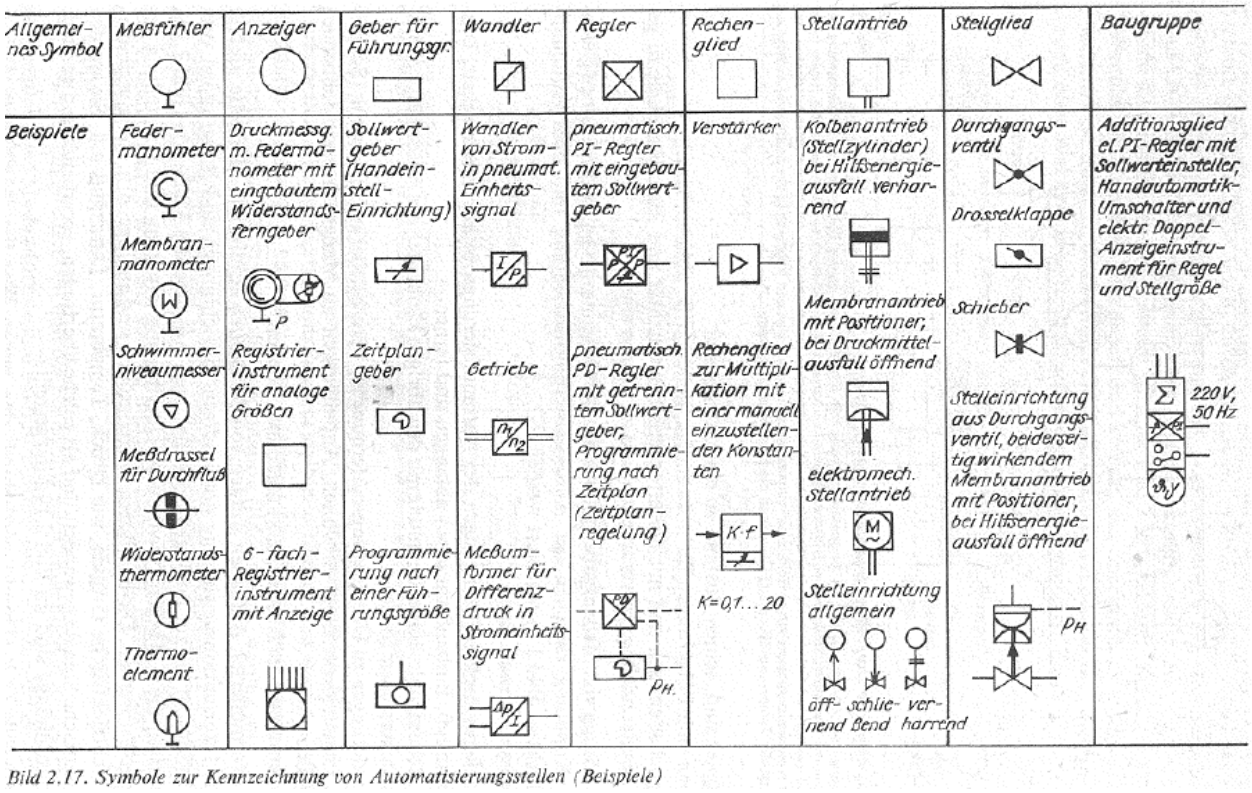


Bild 2.17. Symbole zur Kennzeichnung von Automatisierungsstellen (Beispiele)

Abbildung 3.5: Schaltsymbole
aus ([TB87])

Blockdiagramme

Viele Anwendungsgebiete benutzen Blockdiagramme zur Beschreibung von Systemen. Blockdiagramme sind nach [Sch97b] anschauliche Beschreibungen des strukturmäßigen Aufbaus von Baugruppen. Nach [Kar95] werden Blöcke aus Basiskomponenten und Benutzer/innendefinierten Komponenten zusammengestellt.

In der Signalverarbeitung beispielsweise sind Basiskomponenten z. B. Signale, Benutzer/innendefinierte Komponenten beschreiben Operationen auf den Signalen sowie ihre Ein- und Ausgänge. Blöcke können wiederum Basisblöcke oder zusammengesetzte Blöcke sein.

Abbildung 3.6 zeigt ein Blockschaltbild eines GPS Empfängers (Global Positioning System, wird z. B. für Fahrtenleitsysteme verwendet) [Hir98].

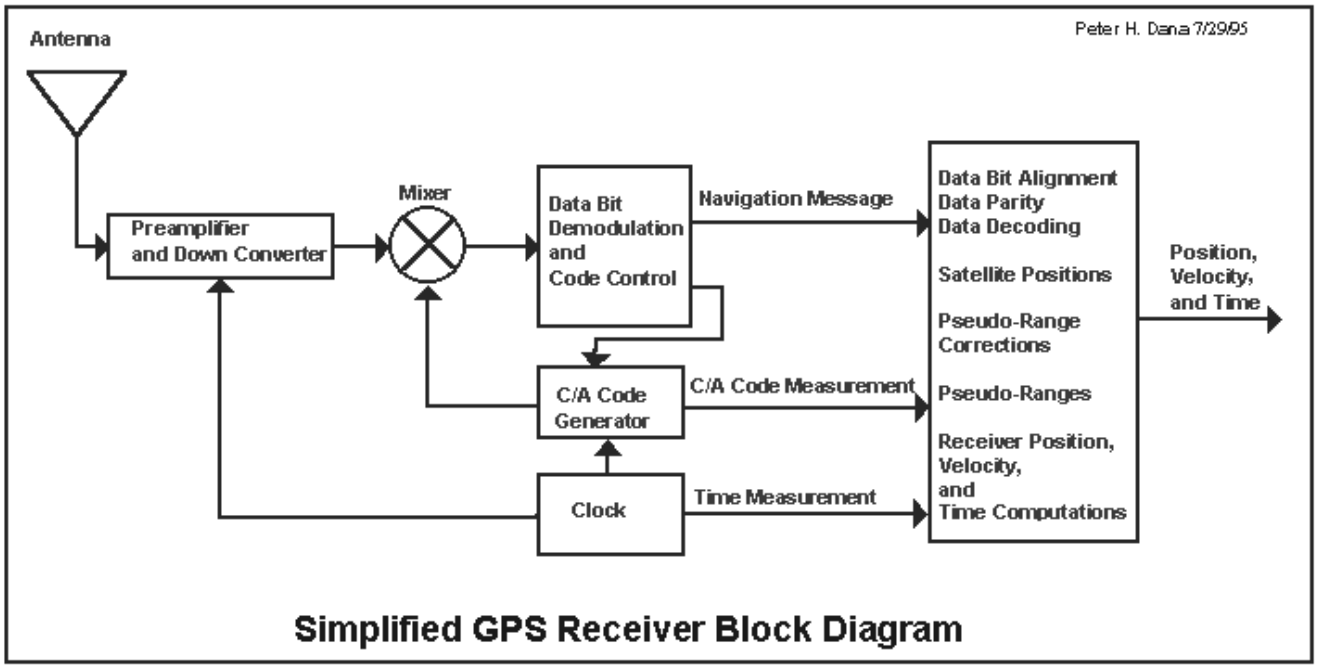


Abbildung 3.6: Blockdiagramm für einen GPS-Empfänger

Mehrere Darstellungen

Die gesamte Anlage wird in mehreren Diagrammen beschrieben, wie in Abbildung 3.7, der Darstellung einer pneumatischen Steuerung, zu sehen ist. Dabei werden Block- und Flußdiagramme verwendet, sowie mathematische Darstellungen und eine Schnittzeichnung.

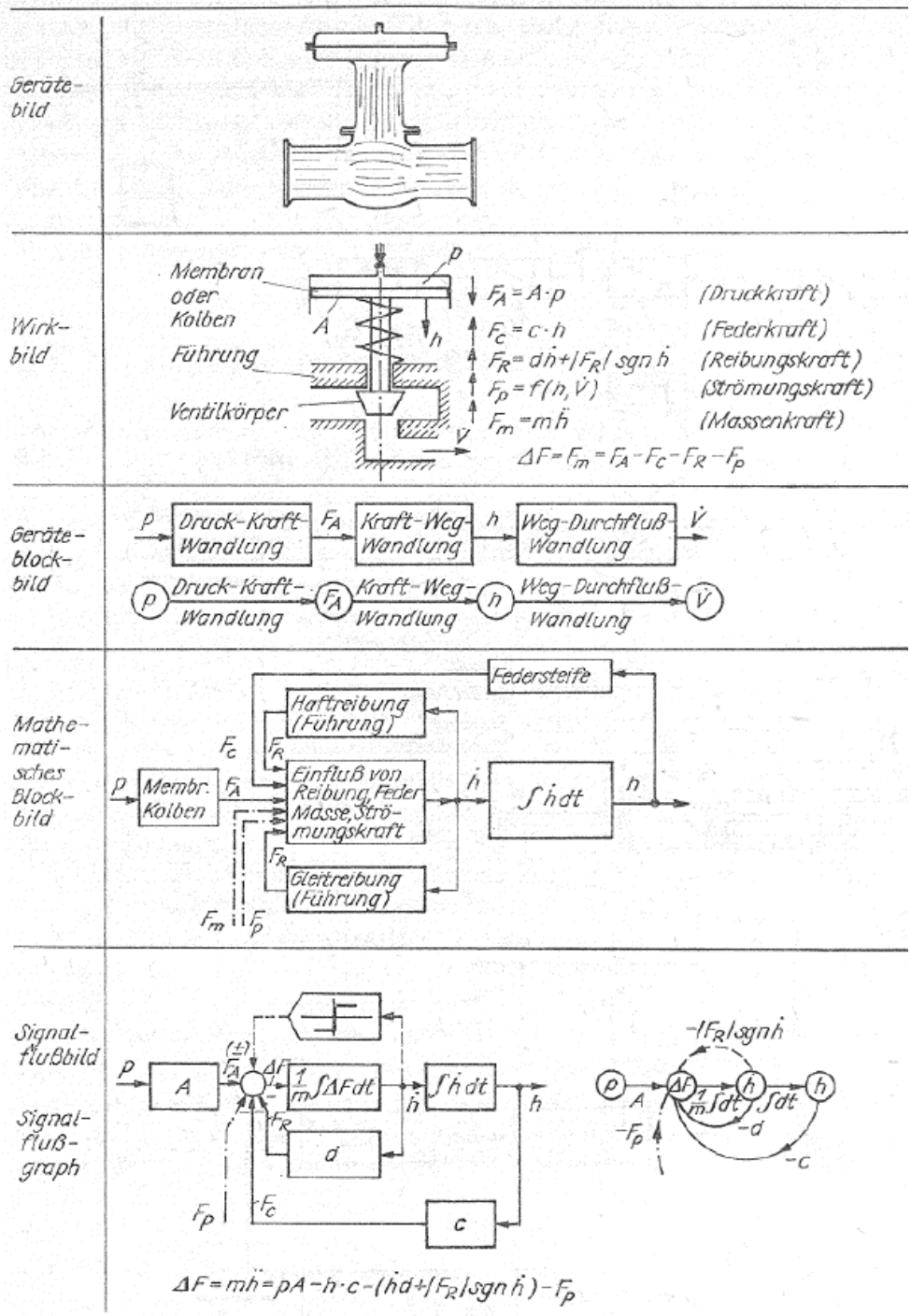


Bild 2.13. Stelleinrichtung mit pneumatischem Antrieb in unterschiedlichen Darstellungen

Abbildung 3.7: mehrere Darstellungen eines Ventils
aus ([TB87])

3.3.5 Untersuchung 2: Einsatz anwendungsspezifischer Entwurfsmethoden und -werkzeuge

Zur Überprüfung der Erkenntnisse aus den vorhergehenden Abschnitten und als Grundlage zur Entwicklung eines unterstützenden Werkzeugs wurde eine Studie durchgeführt, die verschiedene Fragen klären sollte:

- Welche Werkzeuge und Methoden werden beim Entwurf von Benutzungsschnittstellen für Automatisierungssysteme bereits eingesetzt und an welchen Stellen wird weitergehende Unterstützung benötigt?
- Benutzen Entwickler/innen aus verschiedenen Anwendungsdomänen unterschiedliche Werkzeuge und Methoden? Wenn ja, warum? Können anwendungsspezifische Methoden in ein Werkzeug integriert werden?
- Werden die bereits existierenden Werkzeuge überhaupt verwendet? Wenn nicht, warum nicht? Diese Frage führte auch zu einer weiterführenden Studie (siehe Abschnitt 4.5.2).
- Wie erklären die Entwickler/innen die Benutzungsschnittstellen (siehe Abschnitt 2.1.3)? Können aus der Erklärungsweise Entwicklungsmethoden abgeleitet werden?
- Wie in Abschnitt 3.3.4 gezeigt, benutzten Ingenieur/innen zum Entwurf von Automatisierungssystemen mehrere zweidimensionale visuelle Beschreibungen des Systems. Kann diese visuelle Vorgehensweise bei der Entwicklung der Benutzungsschnittstellen wiedergefunden werden? Oder in den Erklärungen?

Insbesondere die Beschäftigung mit der zweiten Frage versprach interessante Ansätze, wie anwendungsspezifisches Wissen in ein Werkzeug integriert werden kann. Durch die Idee, allgemeingültige graphische Spezifikationsmethoden zu verwenden, könnte Wissen, das bei den Entwickler/innen aus verschiedenen Anwendungsbereichen vorhanden ist, integriert werden, ohne dadurch das Werkzeug auf ein spezielles Anwendungsgebiet zu beschränken.

These der Untersuchung

Die Hypothese der Untersuchung war, daß es auf der einen Seite eine Kluft bei der Benutzung von Werkzeugen zwischen den angebotenen (fortschrittlichen und benutzungsfreundlichen) und tatsächlich genutzten (aus der Sicht der Forschung überholten und benutzungsunfreundlichen) Methoden gibt, wie vor der Studie bereits mehrfach beobachtet.

Diese Kluft entsteht zum einen durch den unterschiedlichen Hintergrund der Ausbildung der verschiedenen Entwickler/innengruppen und zum anderen aus der Unfähigkeit der bestehenden Werkzeuge, die Anforderungen sowohl mehrer Anwendungsgebiete als auch der verschiedenen Entwickler/innengruppen zu treffen.

Die Studie

Die Studie basiert auf Interview- und Fragebogentechnik. Testpersonen waren Informatiker/innen und Ingenieur/innen, die Benutzungsschnittstellen für Automatisierungssysteme entwickelt hatten.

Während der Studie wurden acht Entwickler/innen von Benutzungsschnittstellen befragt und beobachtet.

Als Resultate wurde Aufschluß erwartet über

1. die Vielfalt der Methoden, die von Entwickler/innen eingesetzt werden, die nicht Expert/innen auf dem Gebiet der Benutzungsschnittstellen-Entwicklung sind,
2. typische Strukturen von Benutzungsschnittstellen in technischen Anwendungsgebieten,
3. typische Probleme, die auftauchen, wenn Benutzungsschnittstellen in Automatisierungssystemen verwandt werden und
4. Probleme in der Kommunikation zwischen Ingenieur/innen und Informatiker/innen.

Auswahl der Testpersonen

Die Auswahl der Testpersonen wurde mit Hilfe folgender Kriterien getroffen:

- Die Person sollte mindestens eine Benutzungsschnittstelle für ein Automatisierungssystem erstellt haben. Falls sie in einer Gruppe gearbeitet haben sollte, sollte die Gruppe nicht aus mehr als drei Personen bestanden haben, da sonst die Übersicht über das gesamte Projekt nicht gegeben wäre und die Beurteilung der Werkzeuge und Methoden zu stark auf anderen Faktoren beruht hätte.
- Die Person sollte entweder von einer Universität oder aus einer industriellen Umgebung kommen.
- Die Person sollte entweder Informatik, Elektrotechnik oder ein angelehntes Fach studiert haben, damit der Ausbildungshintergrund auf zwei Fächer beschränkt bleibt. Falls die Person Informatik studiert haben sollte, sollte sie auch Erfahrungen im Bereich Automatisierungs- oder Informationssysteme (im technischen Sinne) haben.
- Die Person sollte sich im Bereich HCI auskennen und die Auswahl zwischen verschiedenen Werkzeugen und Methoden gehabt haben.

Die Zugehörigkeit der Testpersonen zu bestimmten Entwickler/innengruppen war wie folgt:

Kriterium	Informatiker/innen	ElektroIngenieur/innen
Universität	2	2
Industrie	1	3

Die untersuchten Benutzungsschnittstellen

Untersucht wurden Benutzungsschnittstellen aus den folgenden Teilbereichen:

- Robotersteuerung (1)
- Dieselmotorsteuerung und -überwachung (2)
- Bildverarbeitung (1)
- Telefonanlagensteuerung (1)
- konfigurierbare Prozeßleitsysteme (3)

Die Befragungsprozedur

Die Befragungsprozedur bestand aus einer Kombination von informellen Interviews und der formalen Beantwortung von formulierten Fragen.

Zusätzlich beobachtete die Interviewerin die Erklärungen, inklusive der Gesten der Testpersonen.

Die Aufzeichungen aus dem Interview und die Skizzen, die die Testpersonen verfertigt hatten, dienten als Auswertungsunterlagen.

Die Interviews wurden in der Arbeitsumgebung der Testpersonen abgehalten und dauerten in der Regel zwei Stunden. Daher konnten die Benutzungsschnittstellen der Interviewerin auch vorgeführt werden.

Das Interview enthielt folgende Fragen:

Fragen, die den Testpersonen gestellt wurden

1. Angaben zur Person
2. Arbeitsbereich
3. Zur Anwendung: Wie war der Aufbau der Anwendung und welches war die Schnittstelle zurBenutzer/in (welche Daten müssen ausgegeben werden)?
4. Beschreiben Sie das Aussehen der Benutzungsschnittstelle:
   a) Beschreiben Sie (mit Skizze oder Ausdruck), wie die drei wichtigsten Bildschirme der Benutzungsschnittstelle aussehen.
   b) Beschreiben Sie, wie die Benutzungsschnittstelle idealerweise aussehen sollte.
5. Welche Oberflächenelemente enthielt die Benutzungsschnittstelle?
   - Buttons (Anzeige ja/nein)
   - Menüs (zur Auswahl von verschiedenen Optionen)
   - extuelle Darstellung von Daten
   - Tabellen
   - Graphische Anzeige von Daten, z. B. Funktionsgraphen, Balkendiagramm
   - Dynamische Anzeige von Daten, z. B. zur Grenzwertüberwachung
6. Welche dieser Elemente sollten idealerweise enthalten sein, wurden aber nicht implementiert. Warum nicht?
7. Was muß die ideale Benutzungsschnittstelle sonders auszeichnen?
8. Was ist die ideale Vorgehensweise, wie sollte das Entwerfen funktionieren?
9. Innerhalb welcher Metaphern wurde beim Entwurf gedacht?
10. Welche graphischen Methoden wurden innerhalb der Metaphern benutzt?
11. Welche Metaphern werden in der Benutzungsschnittstelle selbst verwendet?
12. Welches Tool haben/wollen Sie benutzen. Warum?
13. Welches Tool haben Sie nicht benutzt. Warum?
14. Welches würden Sie gerne benutzen. Warum?
15. Wer hilft dabei - gibt es ein Entwicklerteam?
16. Wie lange haben Sie für die folgenden Phasen gebraucht:
    a) Planung?
    b) Entwurf?
    c) Programmierung?
    d) Verbindung mit der Anwendung?
    e) Testen?
17. Welche Probleme gab es bei der Entwicklung der Benutzungsschnittstelle?
18. Was sind Ihrer Meinung nach die gravierenden Unterschiede in der Vorgehensweise zwischenInformatiker/innen undIngenieur/innen?

Zusätzlich wurde *beobachtet,* welche Methoden die Testperson beim Erklären der Benutzungsschnittstelle benutzte.

3.3.5.1 Ergebnisse der Studie

Benutzte Methoden und Rückschlüsse auf das konzeptuelle Modell

Alle Teilnehmer/innen erklärten die Benutzungsschnittstelle mithilfe folgender Elemente:

- einer Skizze der Benutzungsschnittstelle
- einer Skizze der Anwendung
- einer Skizze der Ojektstruktur der Anwendung
- Verbindungspfeilen, die die Abhängigkeiten zwischen den drei Skizzen ausdrückten
- gesprochener Erklärungen
- deklarativer Spezifikationen, mehrerer visueller Darstellungen und graphischer Interaktion.

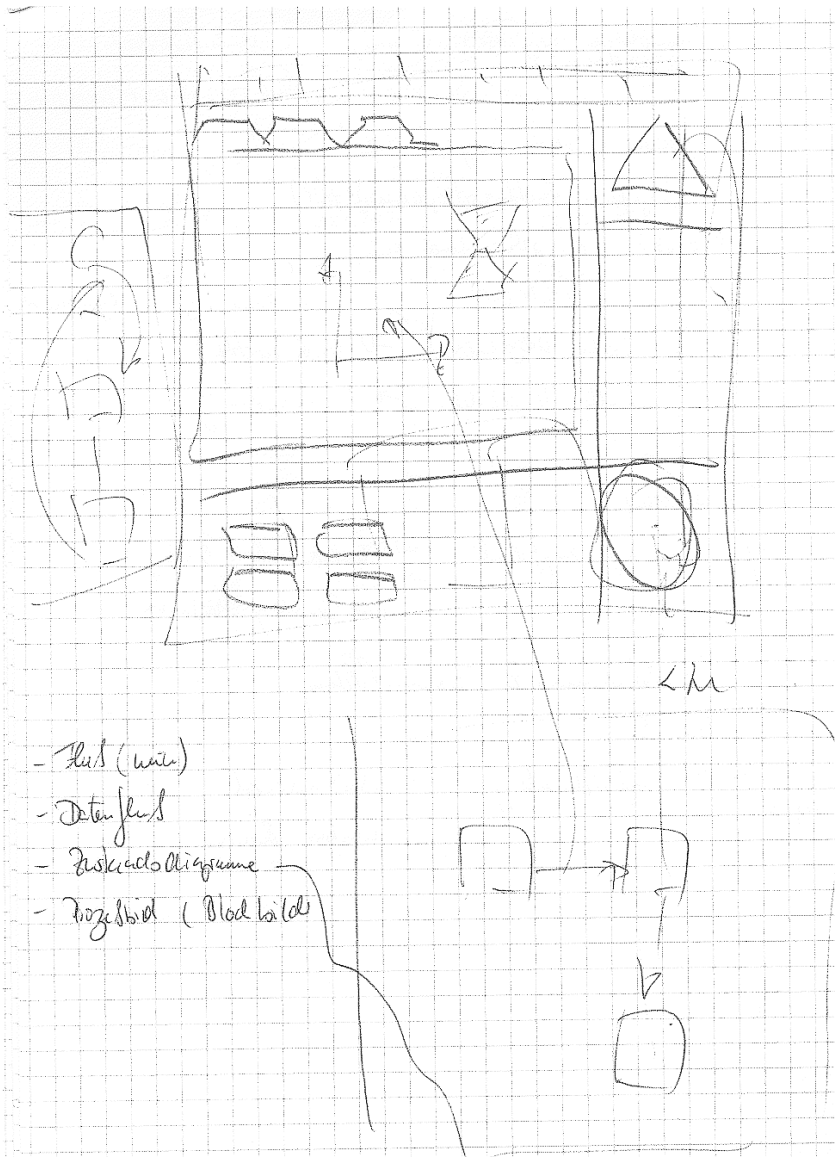


Abbildung 3.8: Skizze als Erklärung zu einer Benutzungsschnittstelle

Die Methoden, die sie tatsächlich benutzten, boten jedoch nur einen Teil dieser Elemente:

- Fünf Personen benutzten über die rein textuelle Programmierung hinaus nur sog. *Interface Builder* (siehe Abschnitt A.2).
- Zwei Personen benutzten darüber hinaus auch *Zustandsdiagramme* zum Entwurf, eine davon mit Werkzeugunterstützung.

Erstaunlicherweise benutzte keine der Testpersonen objektorientierte Entwurfsmethoden.

Kapitel 6 greift die aus den Erklärungen beobachteten Erkenntnisse auf und untersucht, welche visuellen Methoden zur Beschreibung von Benutzungschnittstellen für Automatisierungssysteme geeignet sind.

Typische Anordnungsstrukturen

Alle von den Testpersonen erstellten Benutzungsschnittstellen bestanden aus folgenden Elementen: einem *Kontrollrahmen* sowie einem *Arbeitsfenster* für die *Visualisierung der Anwendung*. Diese Visualisierung wurde mit Hilfe spezieller Werkzeuge für Computergraphik oder den graphischen Bibliotheken des Fenstersystems bzw. der Programmiersprache erstellt.

Auf solchen typischen Anordnungen baut die Mustersprache in Abschnitt 4.4.2 auf.

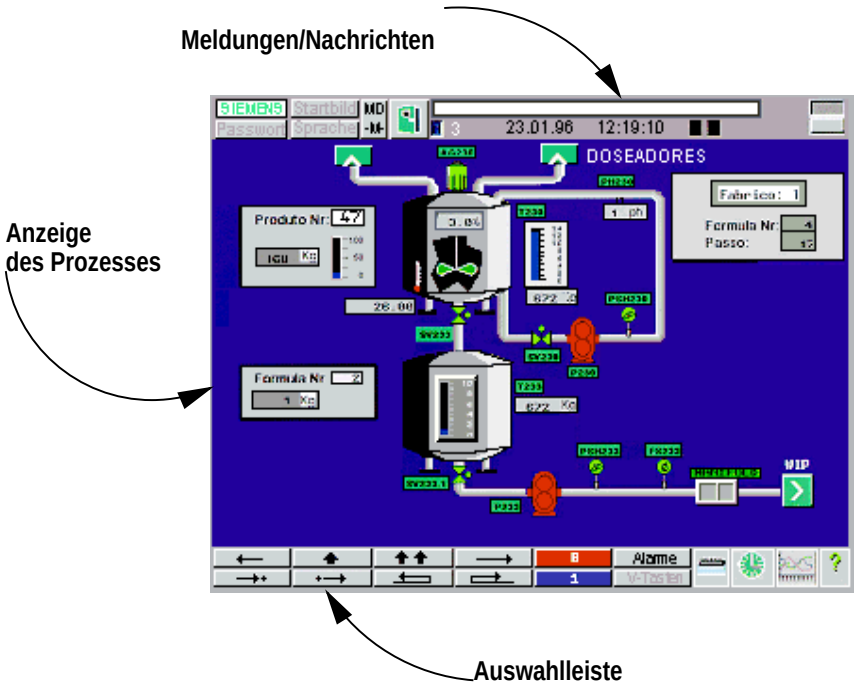


Abbildung 3.9: Typische Anordnung der graphischen Benutzungsschnittstellen-Elemente einer technischen Benutzungsoberfläche

Ein übliches Vorgehen ist auch, graphisches Verhalten durch wechselseitiges Anzeigen von kleinen Bildern zu simulieren, z. B. kann die drehende Pumpe in der Abbildung durch drei Bildchen des Umrührers dargestellt werde, die jeweils eine andere Position zeigen.

Kapitel 4

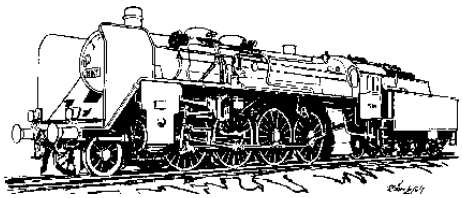
Konstruktion von Benutzungsschnittstellen

In den letzten beiden Kapiteln wurde vorgestellt, welche Fähigkeiten Entwickler/innen von Benutzungsschnittstellen brauchen und welche besonderen Anforderungen Benutzungsschnittstellen technischer Systeme an die Entwickler/innen stellen. In diesem Kapitel werden die ***technischen Grundlagen für die Erstellung objektorientierter Benutzungsschnittstellen*** vorgestellt.

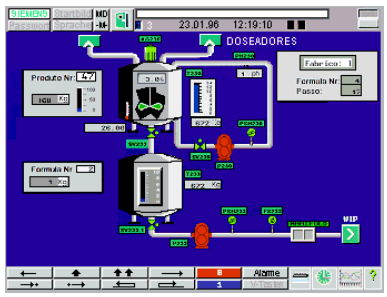
Viele Lehrbücher über Benutzungsschnittstellen beschreiben deren Erstellung aus der Sicht der HCI-Expert/innen. Informatik-Lehrbücher beschränken sich oft auf die Darstellung im Rahmen einer Programmiersprache oder Klassenbibliothek; Methoden wie Entwurfsmuster oder Komponententechnologie werden zumeist vernachlässigt.

Eine Entwurfsmethode, die das konzeptuelle Modell der Entwickler/in berücksichtigt und für viele Entwickler/innen anwendbar ist, muß aber auf einer Darstellung beruhen, die unabhängig von einer bestimmten Sprache oder Programmierumgebung ist. Deshalb werden in diesem Kapitel die im Rahmen der vorliegenden Arbeit gemachten Erfahrungen mit dem Umgang verschiedener Sprachen, Systeme und Entwurfsmethoden zusammengefaßt. Dabei werden die verschiedenen technischen Modelle deutlich, die die konzeptuellen Modelle der Entwickler/innen beeinflussen. Die sprach- und umgebungsübergreifende Zusammenfassung ist Grundlage für ein ingenieurtechnisches Vorgehen zur Konstruktion von Benutzungsschnittstellen, wie im folgenden Vergleich beschrieben wird.

Ingenieurtechnisches Vorgehen: Ein Vergleich



Um eine *Maschine*, beispielsweise die nebenstehende Lokomotive, *konstruieren* zu können, muß man wissen, aus welchen Teilen sie besteht, wie diese Teile zusammengesetzt funktionieren und wie sie daher prinzipiell zusammengesetzt werden müssen. Dann kann man auch Werkzeuge und Maschinen für die Herstellung der einzelnen Teile konstruieren.



Um eine *Benutzungsschnittstelle*, wie die im letzten Kapitel vorgestellte komplexe Tank-Benutzungsschnittstelle, *konstruieren* zu können, muß man ebenso wissen, aus welchen „Teilen“ Benutzungsschnittstellen bestehen. Es wird gezeigt, wie sie prinzipiell funktionieren und wie sie zusammengesetzt werden können.

Um ein solches ingenieurtechnisches Vorgehen zu ermöglichen, sollen folgende Fragen beantwortet werden:

- Was ist eine Benutzungsschnittstelle?
- Woraus besteht eine Benutzungsschnittstelle?
- Was sind die verschiedenen Architekturmodelle von Benutzungsschnittstellen?
- Wie funktionieren Benutzungsschnittstellen in den unterschiedlichen Architekturmodellen?
- Wie wird die Entwicklung von Benutzungsschnittstellen über die Basisprinzipien hinaus unterstützt durch
  - Methoden und Prinzipien des Software Engineering?
  - Werkzeuge?

Was ist eine Benutzungsschnittstelle?

Eine *Benutzungsschnittstelle* ist das Bindeglied zwischen Benutzer/in und Anwendungsprogramm, das den Informationsaustausch steuert (nach [Sch97b, HBvB+94]).

Dazu gehören die *Ein- und Ausgabegeräte*, das *Dialogverhalten* und die *Ein- und Ausgabegestaltung* des Systems.

Die Benutzer/in versteht unter der Benutzungsschnittstelle die *Benutzungsoberfläche*, also das, was von diesem Bindeglied erfahrbar ist.

Die Programmierer/in versteht unter der Benutzungsschnittstelle alle *Programmteile*, die

- die Ein- und Ausgabegeräte kontrollieren,
- die Ausgabetechnik (z. B. Fenstertechnik) zur Verfügung stellen,
- die Daten vom Anwendungsprogramm entgegennehmen, auswerten, aufbereiten und der Ausgabe zur Verfügung stellen und
- die Eingaben der Benutzer/in entgegennehmen und dem Anwendungsprogramm zur Verfügung stellen.

In dieser Arbeit wird die Hardware der Benutzungsschnittstelle nicht betrachtet, und die Benutzungsschnittstellen-Software aus dem Blickwinkel der Programmierer/in untersucht.

Benutzungsschnittstellen, deren Ausgaben zweidimensionale graphische Oberflächenelemente (und Text) sind, und mit denen die Benutzer/in über die Manipulation dieser Elemente (und der Eingabe von Text) interagiert, werden *graphische Benutzungsschnittstellen (Graphical User Interfaces, GUI),* manchmal auch *WIMP-Benutzungsschnittstellen* (d. h. bestehend aus *w*indows, *i*cons, *m*enus und *p*ointing devices) genannt.

Graphische Benutzungsschnittstellen sind oft objektorientiert realisiert; der Begriff *objektorientierte Benutzungsschnittstelle* wird aber in der Literatur unterschiedlich interpretiert: Eine

Benutzungsschnittstelle, die objektorientiert implementiert ist, d. h. durchgehend aus Objekten besteht, wird objektorientiert genannt. Ebenso werden Benutzungsschnittstellen, bei denen sich die Oberflächenelemente wie Objekte verhalten, die aber z. B. prozedural implementiert sind [Jos94] objektorientiert genannt.

Ein Oberflächenelement wird als objektorientiert bezeichnet, wenn es als unabhängige Einheit auf dem Bildschirm dargestellt wird, einen veränderbaren Zustand hat und Nachrichten empfangen und auslösen kann [Ols98].

4.1 Schichten graphischer Benutzungsschnittstellen

„Die Benutzungsschnittstelle" wird zunächst durch eine Spezifikation (z. B. Algorithmen) und dann durch ein Programm beschrieben und „existiert" zur Laufzeit als eine Menge von Prozessen, d. h. im Fall der objektorientierten Benutzungsschnittstellen als Menge von interagierenden Objekten, die auch die Dienste von Basisprogrammen in Anspruch nehmen.

Die folgende Abbildung zeigt die verschiedenen aufeinander aufbauenden Schichten:

Abbildung 4.1: Die Schichten von Benutzungsschnittstellen. Die Entwicklung der eingerahmten Schichten ist Gegenstand dieses Kapitels

Grundlage ist das *Betriebssystem* sowie andere Basisprogramme.

Die *graphischen Treiber* sorgen für die Hardware-Verbindung, und die *Graphikschicht* stellt die Funktionen zur Ausgabe von graphischen Elementen und Text zur Verfügung.

Das *Fenstersystem (*engl. *windowing system)* ist dafür verantwortlich, die Bildschirmausgabe zu verwalten und in mehrere Fenster aufzuteilen.

Auf dieser Basisfunktionalität bauen die *Oberflächenelement-Verwaltung* und die *Widgets* (ein Kunstwort aus window und gadget (Ding), eine deutsche Übersetzung ist etwa *Bausteine*) sowie die *Menge der Fensterinhalte* auf. Die Widget-Schicht wird auch *Toolkit* (Baukasten) [Mye96] genannt, genauso wie die Beschreibung der Widgets. Dazu gehören Bausteine wie Knöpfe (engl. *buttons*), Rollbalken (engl. *scrollbars*), Eingabefelder oder anwendungsspezifische Bausteine wie Funktionsgraphen in der Mathematik oder technische Elemente in der Elektrotechnik.

Die *Benutzungsoberfläche*, also die Präsentation für die Benutzer/in, ist aus Widgets und Fensterinhalten zusammengesetzt.

Die *Konstruktion einer konkreten Benutzungsschnittstelle* erfolgt durch das Erstellen einer Beschreibung der Elemente; je nach Entwicklungsphase handelt es sich bei dieser Beschreibung um Skizzen, Spezifikationen oder, in der letzten Entwurfsphase, um Programme.

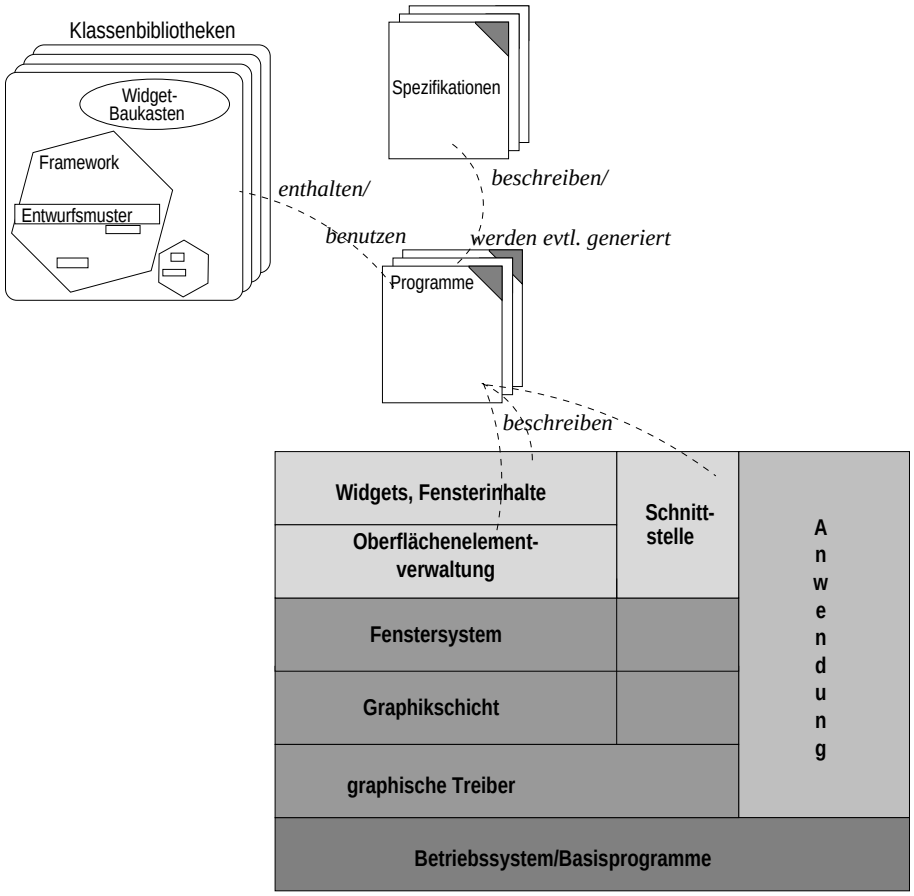


Abbildung 4.2: Konstruktion einer Benutzungsschnittstelle

In der Regel werden nur die obersten Schichten beschrieben und programmiert, d. h. Fenstersysteme und Graphikschicht werden, wie das Betriebssystem, als gegeben angenommen. Was im einzelnen programmiert werden muß, hängt von den Voraussetzungen ab:

- Sind passende Widgets vorhanden, müssen Präsentation und Verhalten der Benutzungsschnittstelle aus ihnen festgelegt werden:
  - Die Oberfläche wird aus ihnen zusammengesetzt (Präsentation),
  - ihr Zusammenspiel wird programmiert (Verhalten),
  - ihre Reaktion auf Eingaben wird programmiert, soweit es über das interne Verhalten hinausgeht (Verhalten),
  - die Art der Ausgabe wird programmiert (Verhalten),
  - und die Anbindung zur Anwendung (z. B. das Anstoßen von Aktionen in der Anwendung aufgrund von Benutzer/inneneingaben oder die Darstellung von Werten der Anwendung in einem Fenster) programmiert (Verhalten).

- Fehlen Widgets, müssen diese programmiert werden (Aussehen und internes Verhalten).

Die Programmierung des Verhaltens ist ein Schwerpunkt dieser Arbeit; Bewertungen der erzeugten Benutzungsschnittstellen, z. B. Bewertung einer erzeugten Präsentation, sind Gegenstand anderer Arbeiten.

Für die Programmierung des Verhaltens muß die Modularisierung bekannt sein (verschiedene Architekturen werden im folgenden Abschnitt vorgestellt), sowie die Mechanismen, die das Verhalten bestimmen (sie werden in Abschnitt 4.3 beschrieben). Zum leichteren Erstellen der graphischen Benutzungsschnittstellen-Elemente und zum Formulieren der Anbindung an das Anwendungs-Programm gibt es Werkzeuge, die auf höheren Abstraktionsebenen arbeiten, die sog. *Higher-Level Tools* [Mye95], wie Abbildung 4.3 zeigt. Eine kurze Beschreibung der Werkzeuge findet sich in Anhang A.

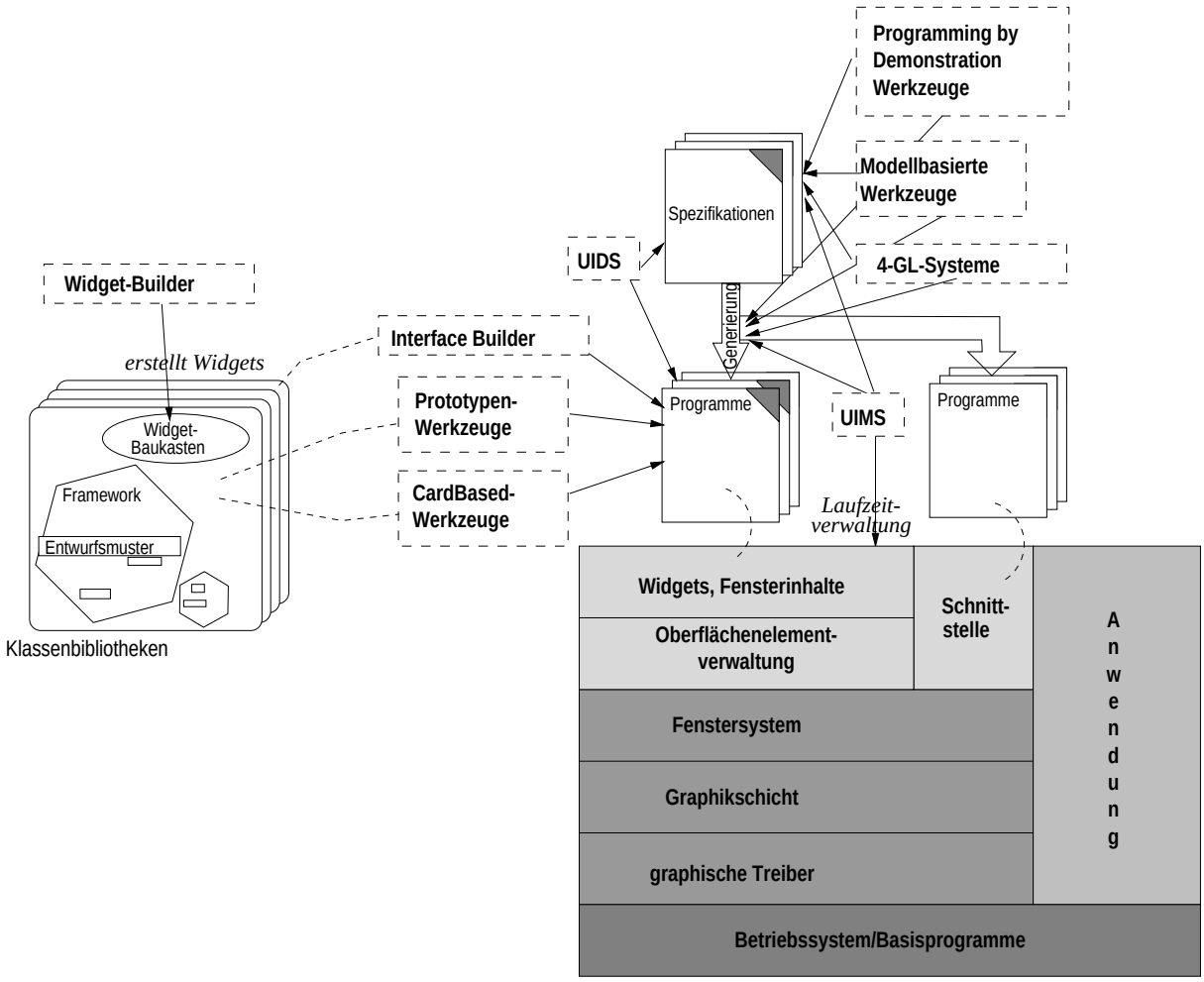


Abbildung 4.3: Werkzeuge zur Erstellung von Benutzungsschnittstellen

4.2 Architekturmodelle für Benutzungsschnittstellen

Architekturmodelle beschreiben die verschiedenen möglichen Bauweisen von Systemen, d. h. der Benutzungsschnittstellen; dabei werden der strukturelle Aufbau, die Modularisierung und

die Verteilung grundsätzlicher Aufgaben auf die einzelnen Module festgelegt (nach [ZZ92]). In der Regel beschreiben solche Modelle die Widget/Fensterinhalts-, Oberflächenverwaltungs- und Schnittstellenschicht (gemäß Abbildung 4.1) im Zusammenhang mit der Anwendung.

Die grundsätzlichen Aufgaben, die in den in der Literatur dargestellten Modellen beschrieben werden, sind: Präsentation, Interaktion mit der Benutzer/in (oft auch Dialog genannt), Verwaltung und Interaktion mit der Applikation (siehe z. B. [ZZ92, Lar92, Ols98, BMR+96]).

Die in einem Architekturmodell vorgeschlagene Strukturierung und Modularisierung kann, abhängig von der Phase des Entwurfs, unterschiedlich interpretiert werden. Diese Interpretation reicht von der reinen Strukturierung der abstrakt formulierten Aufgaben über das Verständnis der einzelnen Subsysteme als Agenten bis hin zu einer Aufteilung des Programmcodes. Daraus resultieren oft Verständnisschwierigkeiten, z. B. wird MVC (siehe unten) in verschiedenen Berichten beschrieben als: MVC-Architektur [Lar92], MVC-Paradigma [KP88, Lar92], MVC-Framework [Par95], MVC-Entwurfsmuster [BMR+96], MVC-Modell [Sch96], MVC-Prinzip [Zha97] oder als ein System kooperierender Agenten [LM00].

Die Bedeutung des Architekturmodells

Das Architekturmodell hat durch seine Modularisierung Einfluß auf die Vorstellung, d. h. das konzeptuelle Modell der Entwickler/in und damit auf den Entwicklungsprozeß. In diesem Abschnitt werden einige bekannte Architekturmodelle, anhand der Aufteilung der Aufgaben auf einzelne Subsysteme, vorgestellt (wie auch immer diese realisiert sind). In Kapitel 5 wird ein weiteres Modell beschrieben, das Aspektmodell, das auf dem Aspektansatz beruht (siehe Abschnitt 2.1.4) und als Grundlage für diese Arbeit entworfen wurde.

Das Zusammenspiel der einzelnen Subsysteme in den verschiedenen Realisierungen wird in Abschnitt 4.3 beschrieben.

Monolithische Architektur

Hier sind alle Aufgaben in einem einzigen Modul realisiert. Diese Architektur wird in neueren Systemen nur sehr selten verwendet, da die enge Verknüpfung die unterschiedlichen Aufgaben zu sehr miteinander verwebt, um sinnvoll wartbar und wiederverwendbar zu sein (siehe z. B. [ZZ92, Lar92]).

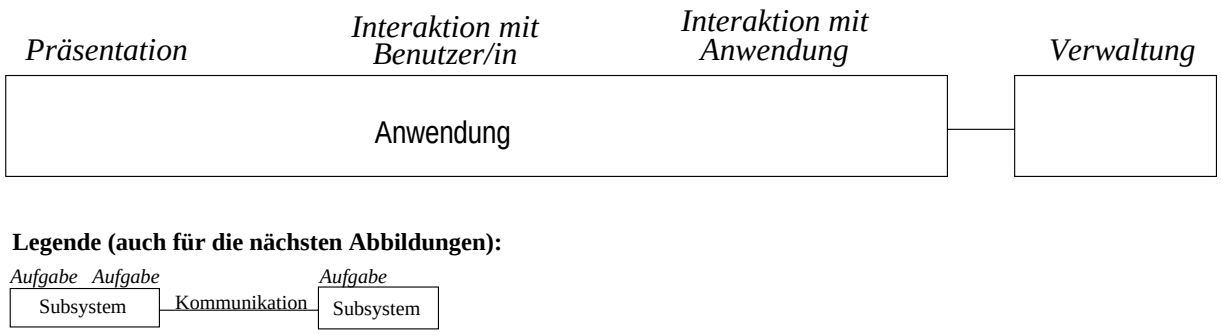


Abbildung 4.4: Monolithische Architektur

Client-Server-Architektur

Hier wird die Widget-Schicht, also die Darstellung auf der Oberfläche, getrennt von den Interaktionsfunktionen gehalten. Ein Beispiel für diese Architektur ist das X Window System. Solche Benutzungsschnittstellen sind objektbasiert, d. h. die Oberflächenelemente verhalten sich wie Objekte, der Client-Teil ist monolithisch und meistens nicht objektorientiert realisiert (nach [ZZ92, Lar92]).

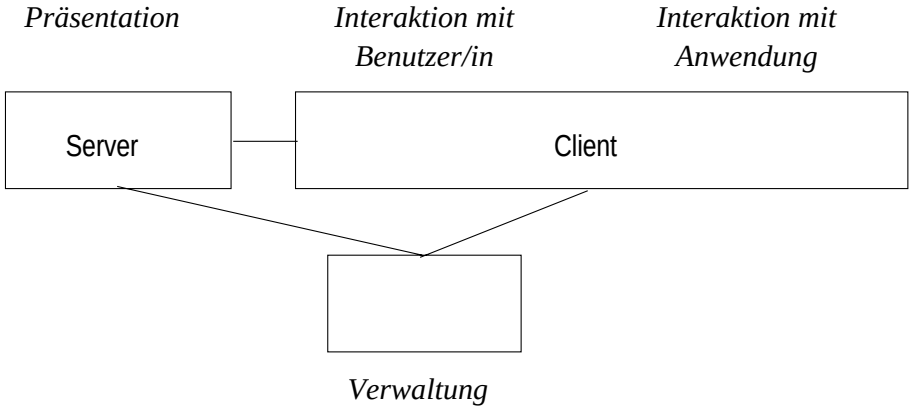


Abbildung 4.5: Client-Server-Architektur

Seeheim-Architektur

Diese Architektur wurde in den 80er Jahren für kommandobasierte Benutzungsschnittstellen entwickelt, bei denen die Interaktion mit der Benutzer/in für das ganze System durch eine Zustandsmaschine beschrieben werden konnte, d. h. in jedem möglichen Zustand des Systems führt die Eingabe der Benutzer/in zu einem Übergang in einen anderen Zustand. Diese Steuerung wird explizit in dem Dialogsteuerungsmodul beschrieben, das getrennt von Präsentation und Anwendung gehalten wird. Für direktmanipulative Benutzungsschnittstellen ist dieses Modell nicht so gut geeignet, da die Anzahl der Zustände durch die beliebigen Manipulationsmöglichkeiten zu groß wird, die Semantik einer Manipulation nicht zu jedem Zeitpunkt eindeutig beschrieben ist (siehe z. B. [Col94]) und mehrere Objekte gleichzeitig manipuliert werden können.

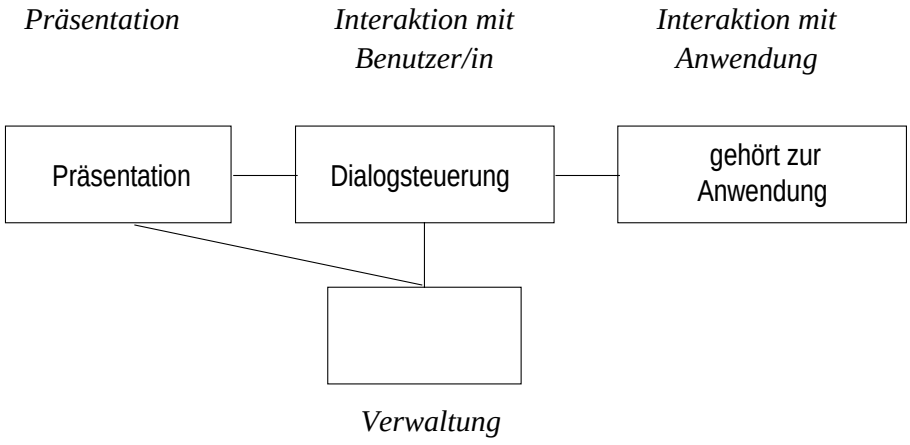


Abbildung 4.6: Seeheim-Modell

Model View Controller-Architektur (MVC)

Die MVC-Architektur ist für objektorientierte Systeme entworfen. Die Trennung von Ausgabe und die Behandlung der Eingaben der Benutzer/in erfolgt auf Objektebene, wobei je ein `View`- und ein `Controller`-Objekt zusammengehören, also die Widget-Schicht durch `View-Controller`-Paare aufgebaut wird. Die Paare werden an je ein `Model`-Objekt gekoppelt, zu jedem `Model` können beliebig viele `View-Controller`-Paare gehören. Da die Widget-Schicht aus kleinen, relativ selbständigen Einheiten aufgebaut ist, spricht man auch von *Mehragentensystemen* [Lar92]. Auch die nächsten beiden Architekturmodelle werden als *Agentenmodelle* bezeichnet.

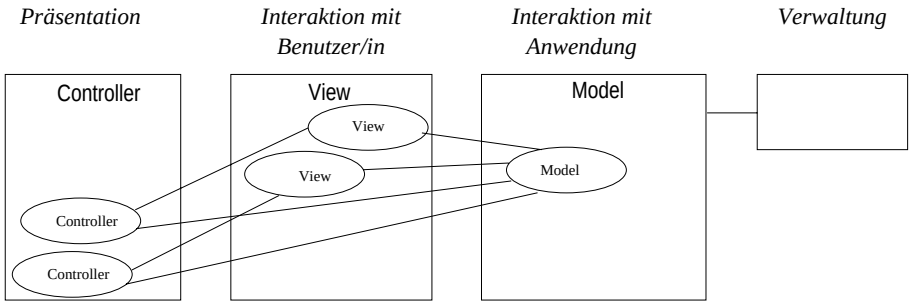


Abbildung 4.7: Model View Controller -Architektur

Document-View Architektur

Bei dieser Architektur werden View und Controller aus der MVC-Architektur zu einem Objekt zusammengefaßt, d. h. die Präsentation übernimmt auch die Verarbeitung der Eingaben [GHJV95].

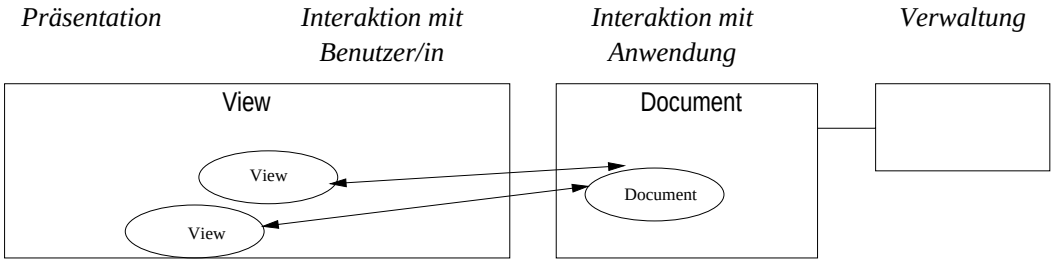


Abbildung 4.8: Document-View Architektur

Presentation Abstraction Control-Architektur (PAC)

Die Benutzungsschnittstelle besteht aus einer Hierarchie von Agenten, die auch die Aggregationshierarchie widerspiegelt [CNS94, BMR+96]. Jeder einzelne Agent besteht aus drei Elementen:

- der Präsentation, die die Ein- und Ausgabe realisiert (also `View` und `Controller` im MVC-Modell, bzw. View im Document-View-Modell),
- der Abstraktion, die die für dieses Oberflächenelement wichtigen Daten enthält und

- dem Kontrollelement, das die Kommunikation mit den anderen Agenten realisiert.

Der Agent an der Wurzel der Hierarchie ist in der Regel für den Zugriff auf die Anwendungsdaten zuständig, der Behälter ist für die restlichen Benutzungsschnittstellen-Elemente zuständig. Die Blätter entsprechen den Widgets und die dazwischenliegenden Agenten erfüllen alle sonstigen Aufgaben, z. B. als Container von Widgets.

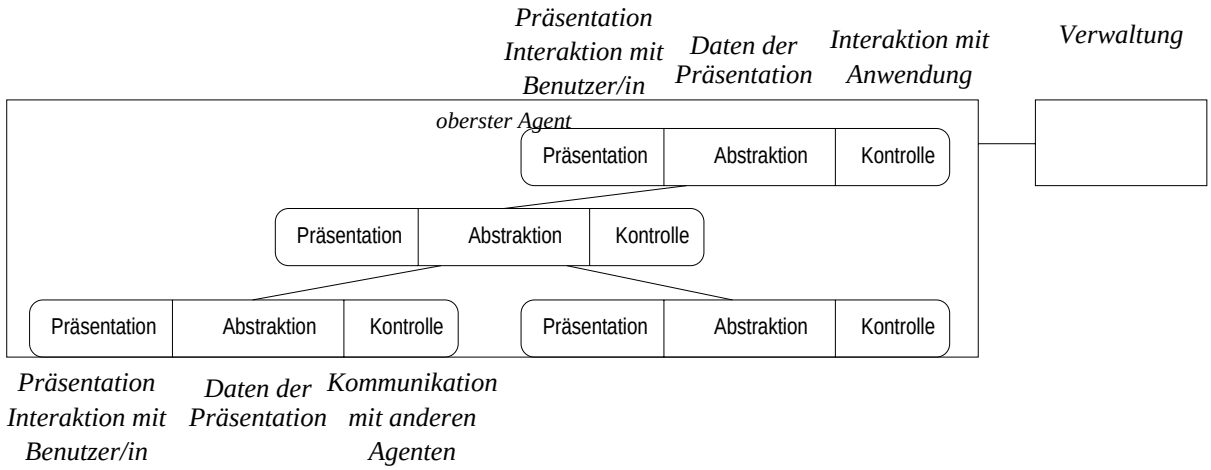


Abbildung 4.9: Presentation Abstraction Control-Architektur

4.3 Verhalten von Benutzungsschnittstellen

Dieser Abschnitt stellt einen Rahmen für die bekanntesten Verhaltens-Mechanismen von Benutzungsschnittstellen dar, mit dessen Hilfe Realisierungen dieser Mechanismen schnell verstanden werden können.

Es werden nur die bekanntesten Konzepte vorgestellt und die verschiedenen Realisierungen sehr kurz behandelt.

Das Verhalten einer Benutzungsschnittstelle wird bestimmt durch die Oberflächen- und Schnittstellenelemente, ihre Zustände und ihre Reaktionsmöglichkeiten auf Eingaben. Es gibt verschiedene Basismechanismen zur Steuerung dieses Verhaltens, im wesentlichen zustandsorientierte Steuerung und ereignisbasierte Steuerung, die im folgenden vorgestellt werden sollen. Die in dieser Arbeit vorgestellten Entwurfsmethoden basieren auf ereignisbasierten Benutzungsschnittstellen. Zustandsbasierte Benutzungsschnittstellen werden nur kurz erwähnt. Aufbauend auf den ereignisbasierten Grundmechanismen, die in einem System gegeben sind [Sch96], werden Ereignisverteilungs- und Ereignisbehandlungs-Mechanismen verwendet, mit denen das Verhalten einer konkreten Benutzungsschnittstelle von der Entwickler/in festgelegt werden kann.

Welcher Mechanismus verwendet wird, bestimmt das konzeptuelle Modell der Entwickler/in über die Programmierung der Benutzungsschnittstelle und damit auch die Spezifikations- und Programmiermethoden, die verwendet werden können.

4.3.1 Basismechanismen

In *kommando- und menübasierten Benutzungsschnittstellen* erfolgen die Eingaben der Benutzer/in sequentiell. Die weitere Bearbeitungsmöglichkeit ist nur abhängig vom aktuellen Zustand der gesamten Benutzungsschnittstelle, deren Verhalten damit als *Zustandsautomat* definiert werden kann [Lar92]. Benutzungsschnittstellen, die nach dem in Kapitel 4.2 beschriebenen Seeheim-Architekturmodell modelliert sind, funktionieren nach diesem Prinzip.

In *zweidimensionalen graphischen Benutzungsschnittstellen* ist eine solche Eindeutigkeit nicht in jedem Fall gegeben und die Verarbeitung der Eingabe auch davon abhängig, welches Oberflächenelement aktiv ist (das wird i. d. R. durch die räumliche Position eines Zeigegeräts bestimmt); evtl. sind das sogar mehrere gleichzeitig. Beispielsweise ist bei Selektion und Bewegen eines Dateipiktogramms noch nicht deutlich, ob die Datei verschoben (d. h. über einem Verzeichnispiktogramm fallen gelassen wird) oder gelöscht werden soll (d. h. über dem Papierkorbpiktogramm fallen gelassen wird).

In solchen graphischen Benutzungsschnittstellen werden daher aufgrund der Eingaben der Benutzer/in *Ereignisse* erzeugt, die Aktionen in der Anwendung anstoßen. Das Ereignis ist (nach [Mye92]) ein Objekt (oder eine andere Datenstruktur), das Information über die von der Benutzer/in vorgenommene Aktion enthält, z. B. Ereignistyp, Zeigerposition und Zeitpunkt [Mye00b]. Dieses Ereignis wird in eine Warteschlange eingereiht und wird, je nach aufgesetztem Mechanismus bzw. Architektur an die entsprechenden Teile der Benutzungsschnittstelle zur Weiterverarbeitung verteilt, siehe Abbildung 4.10.

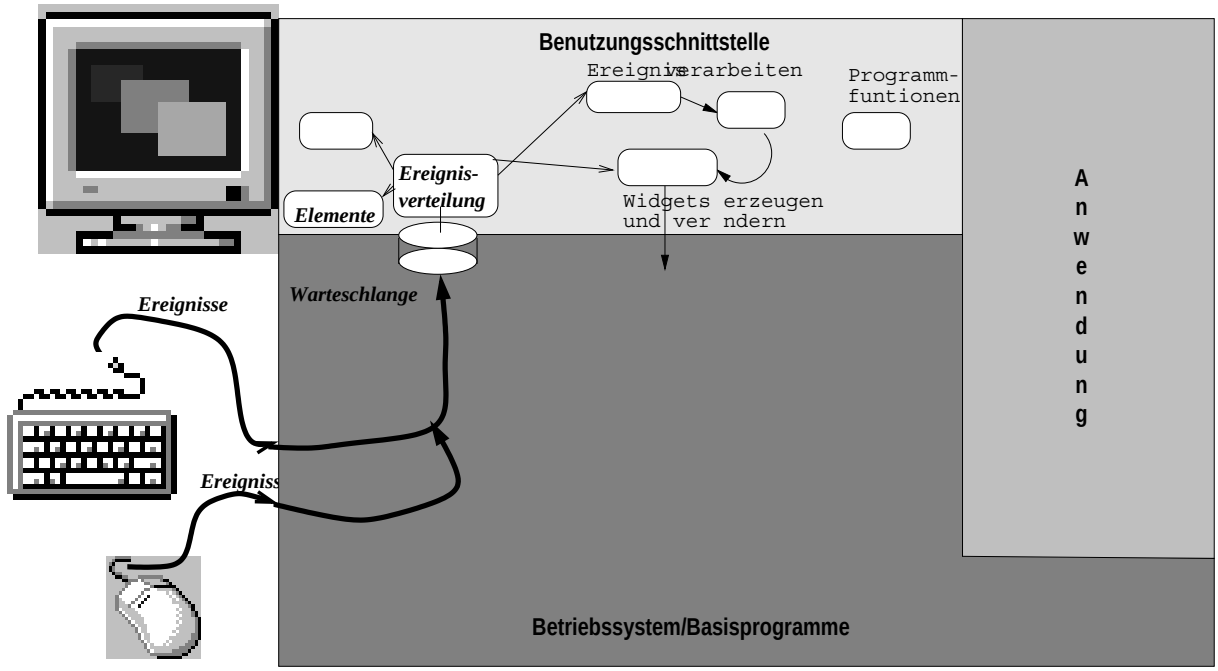


Abbildung 4.10: Ereignisverteilung in einer Benutzungsschnittstelle

Die in Kapitel 4.2 beschriebenen Mehragentenmodelle folgen diesem Prinzip. Um das Verhalten der Benutzungsschnittstelle zu definieren, muß als Basis ein Ereignisverteilungs-Mechanismus verwendet, und darauf basierend ein oder mehrere Ereignisverarbeitungs-Mechanismen implementiert werden.

4.3.2 Ereignisverteilung und Ereignisverarbeitung

Ereignisse sind für die Anwendungssoftware Datenstrukturen:

- In älteren Systemen wurden sie meist durch Konstanten abgebildet, die zur Unterscheidung vorgegebener Fallstrukturen dienten.
- In objektorientierten Systemen werden auch Ereignisse durch Objekte abgebildet.

Ereignisse sind auf unterschiedlichen Abstraktionsniveaus definiert, z. B. ist das Drücken eines einzelnen Tastaturknopfes ein einfaches Ereignis, die Aktion „Markieren einer Textzeile" ist ein Ereignis, das sich aus mehreren Einzelereignissen zusammensetzt (Herunterdrücken des Mausknopfes, Bewegen der Maus über die Zeile, Loslassen des Mausknopfes).

Die Ereignisverteilung geschieht nach einem der folgenden Prinzipien:

Traditionell wurden auftretende Ereignisse innerhalb eines Programmstücks explizit bearbeitet (das Prinzip der *Ereignisschleife*). D. h. nachdem von den unteren Schichten die Aktion einer Benutzer/in an die Benutzungsschnittstelle gegeben wurde, wird ein Ereignis als entsprechende Datenstruktur erzeugt und in dem Programmstück abgearbeitet. In Pseudoprogrammnotation sieht das folgendermaßen aus:

Programmbeispiel: Ereignisschleife

...
while TRUE do

```
begin
   waitForEvent (anEvent)
   case anEvent of:
      anEvent = Event1
        if targetWidget = Widget_1
            begin
              ...  {some code}
            end
        ...
        if targetWidget = Widget_m
            begin
              ...  {some code}
            end
      ...
      anEvent = Event_n
         ...
end
```

...

Bei dem sog. *Callback-Prinzip* wird für jedes Ereignis eine Funktion festgelegt, die bei seinem Auftreten aufgerufen wird ([CS95]) und die dann die Abarbeitung übernimmt.

Programmbeispiel: Callback-Prinzip

```
...
while TRUE do

        begin
            waitForEvent (anEvent)
            case anEvent of:
               anEvent = Event1
                 if targetWidget = Widget_1
                     call function_1
                 ...
                 if targetWidget = Widget_m
                     call function_k
               ...
               anEvent = Event_n
                  ...
         end

...
```

Bei dem *Beobachterprinzip (*engl. *observer principle)*, das in auf Smalltalk oder Java basierenden Systemen verwendet wird, wird die Aktion der Benutzer/in von den unteren Schichten direkt an ein Widget weitergegeben, welches das Ereignis erzeugt. Dabei wird es von vorher festgelegten Objekten beobachtet, die dann die weitere Abarbeitung übernehmen. Dieses Prinzip wird im nächsten Abschnitt genauer erläutert.

Die Festlegung der Ereignisverarbeitung ist bei den Mehragenten-Architekturen eine der Hauptaufgaben der Spezifikation des Verhaltens der Benutzungsschnittstelle und daher grundlegend für das Verständnis der Entwickler/innen. Die Entwicklungssysteme, d. h. die Konzepte der Programmiersprache und die in den *Programmbibliotheken* definierten Funktionsweisen, implementieren die *Syntax* und die *Verteilungs- und Verarbeitungsmechanismen* unterschiedlich und sind daher in der Praxis nicht einheitlich zu benutzen (obwohl sie sich konzeptuell teilweise ähnlich sind).

Die in dieser Arbeit vorgestellten Entwurfsmethoden sind geeignet für den Entwurf von Benutzungsschnittstellen, die auf dem Observer-Prinzip basieren, da dies die neueren Ansätze sind. Um die Entwurfsmethoden zu verstehen, muß dieses Prinzip noch etwas genauer erklärt werden. Der nächste Abschnitt beschreibt daher am Beispiel der bekannten Programmiersprachen Smalltalk und Java, welche Ereignismechanismen durch Sprachkonzepte und die verschiedenen Bibliotheken beschrieben werden.

In der Literatur wird das Observer-Prinzip im Zusammenhang einer konkreten Implementierung vorgestellt; eine Betrachtung mehrerer Implementierungen im Rahmen dieser Arbeit zeigte, daß die Implementierungen das Grundprinzip so sehr aufweichen, daß es oft nur schwer zu erlernen und benutzen ist. Diese Überlegungen bilden auch die Grundlagen für die empirische Untersuchung in Abschnitt 4.5.2, bei der die Visualisierungen des Ereignismodells

verschiedener Entwicklungsumgebungen der beiden Sprachen miteinander verglichen werden. Als Konsequenz wurde in COMBO eine Visualisierung des Observer- bzw. MVC-Prinzips realisiert (auf der Strukturebene, Ebene 2), die implementierungsunabhängig ist (siehe Abschnitt 6.3.4).

4.3.3 Implementierungen von Ereignisbehandlung

Obwohl es viele Werkzeuge gibt, die auf die Entwicklung von Benutzungsschnittstellen spezialisiert sind (siehe Abschnitt A), so daß die Entwickler/in sich eigentlich um die Programmierung der Ereignisbehandlung nicht zu kümmern bräuchte, wird ein großer Teil der technischen Benutzungsschnittstellen „von Hand", d. h. mit Hilfe einer allgemeinen Programmiersprache programmiert.

Zur Zeit wird zur Entwicklung von Systemen - einschließlich der Benutzungsschnittstellen - oft eine integrierte Programmierumgebung, verbunden mit einer Bibliothek aus vorgefertigten Programmstücken (auch *Toolkit* genannt), verwendet. Die Ereignisbehandlung muß aber auf der Strukturebene (Ebene 1) textuell programmiert werden.

Die Bibliothek implementiert die Oberflächenelemente und ihr Verhalten; dadurch kann die Programmiersprache einfach bleiben, d. h. sie hat wenige Sprachelemente und eine einfache Syntax. Dieses Konzept wurde auch bei der Entwicklung der Programmiersprachen Smalltalk und Java eingehalten. Die Sprache Smalltalk wurde gerade im Hinblick auf die Entwicklung von Benutzungsschnittstellen entworfen und wird heute noch verbreitet eingesetzt. Java ist interessant, weil es wichtige Konzepte von Smalltalk und anderen Programmiersprachen aufgegriffen hat und ebenfalls reichhaltige Programmbibliotheken für Benutzungsschnittstellen (z. B. das abstract windowing toolkit (AWT) und die Swing-Programmbibliothek) enthält; Umgebungen für beide Sprachen benutzen (heute) das Observer-Prinzip. Aus einer Betrachtung von Entwicklungsumgebungen dieser beiden Sprachen wurde in dieser Arbeit ein Überblick über aktuelle Konzepte der Benutzungsschnittstellen-Entwicklung erstellt. Als Beispiel wird wieder der Tank diskutiert.

Ereignisbehandlung in Smalltalk-Entwicklungsumgebungen

In den Entwicklungsumgebungen VisualWorks und VisualAge wird das MVC-Modell unterschiedlich implementiert und erweitert:

VisualWorks

stellt die drei Klassen `Model`, `View` und `Controller` zur Verfügung, die über den sog. *Abhängigkeitsmechanismus* (engl. *dependency mechanism*) miteinander kommunizieren:

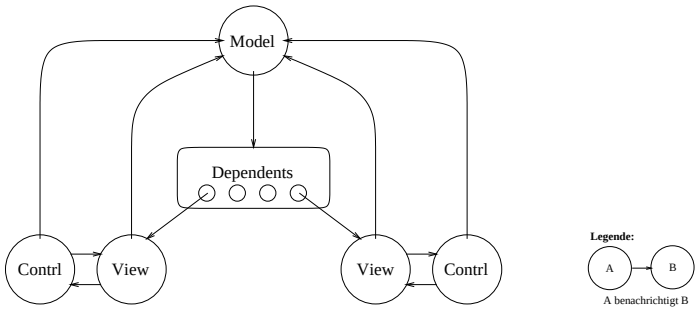


Abbildung 4.11: Die Abhängigkeiten von Model, View und Controller

`View`- und `Controller`-Objekte gehören, wie durch die Architektur definiert, paarweise zusammen und sind jeweils von einem `Model`-Objekt abhängig. Die Ereignisverarbeitung wird durch Ausprägungen der `Controller`-Klasse - genannt `Controller` - realisiert, die das `Model` über die Benutzer/inneneingaben informiert und den `View` gegebenenfalls benachrichtigt. Das Ereignis wird als solches nicht weitergegeben, sondern durch Methodenaufrufe verarbeitet. Jede Komponente der Benutzungsoberfläche hat ihren eigenen `Controller` (wenn sie eine interaktive Komponente ist). Auf der technischen Ebene der Ereignisverarbeitung gibt es zwei Möglichkeiten: Entweder ein `Controller` fragt die Ereignisse ab (sog. *polling*, wird heute seltener benutzt), oder er wird durch den `ControlManager` über das Auftreten eines Ereignisses informiert (ereignisgesteuert).

Hat der `Controller` die Benachrichtung über ein Ereignis erhalten, reagiert er mit Benachrichtigung des `Models` und des `Views`. Wenn das Modell sich ändert, bewirkt der Abhängigkeits-Mechanismus die Benachrichtigung der `Views` und damit indirekt auch der Controller.

In der Regel ist die Anwendung der MVC-Architektur jedoch nicht so einfach wie beschrieben, weil weitere Mechanismen eingeführt wurden, die z. B. der Wiederverwendung dienen. Zur Vereinfachung der Komposition von Elementen in einem Fenster werden das Adapter- und das Decorator-Muster angewandt (siehe Abbildung 4.12, zu Mustern siehe Abschnitt 4.4.2 sowie [GHJV95]), wodurch die `View`-Objekte nicht mehr direkt zugreifbar sind, sondern durch spezielle Adapter- und Dekorationsklassen. Die Kommunikation wird vereinheitlicht und dadurch vereinfacht.

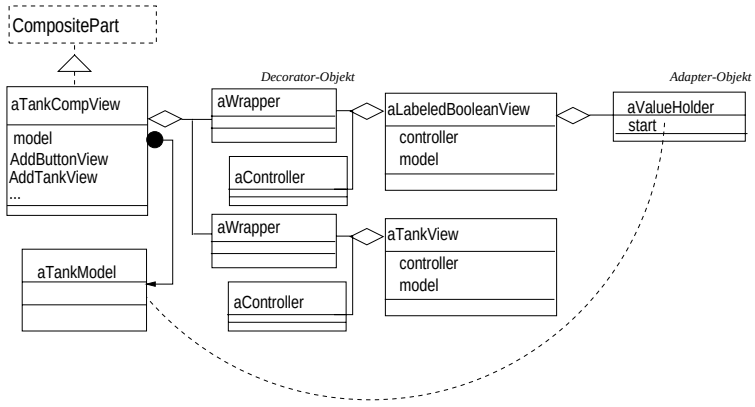


Abbildung 4.12: MVC mit Dekorator (`Wrapper`) und Adapter (`ValueHolder`)

Um die Unabhängigkeit der Oberfläche von der Anwendung weiter zu erhöhen, wurde als

Erweiterung eine weitere Klasse eingeführt: Das `ApplicationModel`, das die `Model`-, `View`- und `Controller`-Objekte, sowie die Verbindung zur Anwendung aus einer Spezifikation, automatisch erzeugt (siehe Abbildung 4.13).

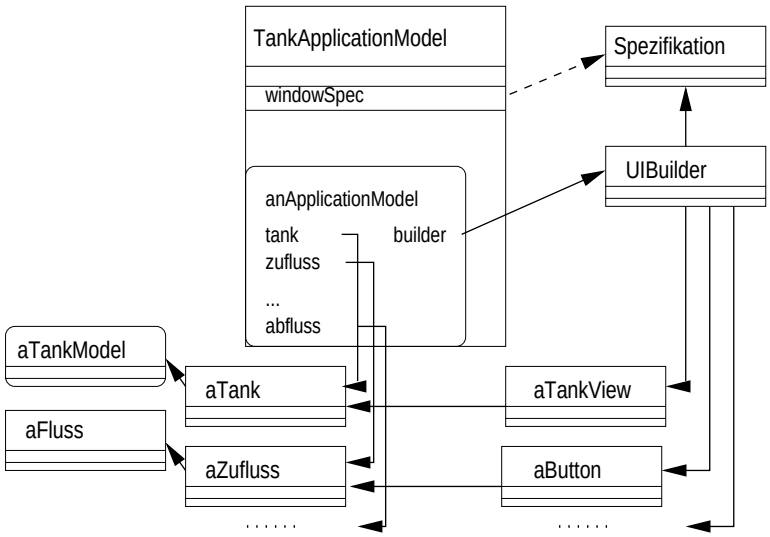


Abbildung 4.13: Erweiterung des MVC-Modells durch die Klasse ApplicationModel

VisualAge

Die Entwicklungsumgebung VisualAge erzeugt Benutzungsschnittstellen nach der Document-View-Architektur. Eine Anwendung wird zentral von einer Application-Klasse verwaltet. Die technische Ebene der Ereignisverarbeitung ist sehr stark abhängig vom benutzten Grafiksystem. Low-Level Ereignisse, wie Tastendrücke und Maus-Tastendrücke, werden durch Ereignisbehandler (engl. *event handler)* bearbeitet, höhere Ereignisse, wie das Schließen eines Fensters, werden durch Rückruf-Funktionen behandelt. Jedem möglichen Ereignis in einem Widget wird eine bestimmte Methode zugeordnet.

Auf diesem Basismechanismus wird ein weiterer Ereignismechanismus definiert, indem Klassen zu Komponenten (siehe Abschnitt 4.4.3) zusammengefaßt werden können, die ebenfalls über Ereignisse kommunizieren.

Ereignisbehandlung in Java-Entwicklungsumgebungen

In dieser Arbeit wurde nur Java in der von der Firma Sun festgelegten Form untersucht. Die Java-Klassenbibliothek wird ständig weiterentwickelt, insbesondere der Teil, der für die Programmierung der Benutzungsschnittstelle zuständig ist. Die verschiedenen Versionen haben jeweils neue Ereignisbehandlungs-Mechanismen.

AWT 1.0 (AWT = abstract windowing toolkit) basierte auf dem Ereignisschleifenprinzip, wie oben verdeutlicht.

AWT 1.1 benutzt das Observer-Prinzip im sog. Event-Delegation-Modell. Im Gegensatz zu Smalltalk, wo die Funktionalität durch Erbung innerhalb der Klassenhierarchie festgelegt wird, muß in Java die Funktionalität ausprogrammiert werden, gemäß den Konventionen einer abstrakt definierten Schnittstelle, den sog. Interfaces.

Die Funktionalität des Observer-Prinzips entsteht durch die Implementierung der abstrakten Schnittstellen `Event-Listener` und `Event-Source`.

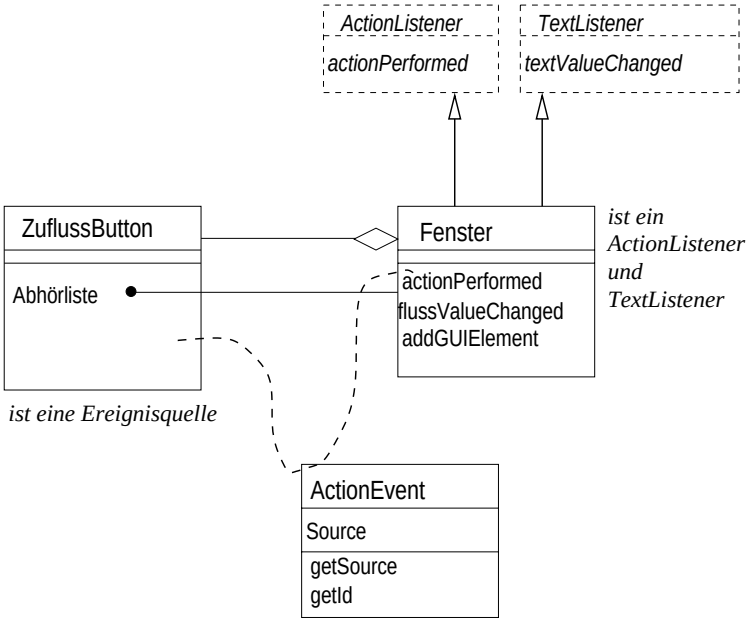


Abbildung 4.14: Ereignisbehandlung im Java AWT 1.1 Anwendungsrahmen

Swing ist ein Anwendungsrahmen, der die Document-View-Architektur realisiert, die aber in der Literatur auch MVC-Architektur genannt wird [WC99, Ost98, Ste97]. Es wird ebenfalls durch die Implementierung abstrakter Schnittstellen, nämlich `Observer` und `Observable`, realisiert, siehe Abbildung 4.15.

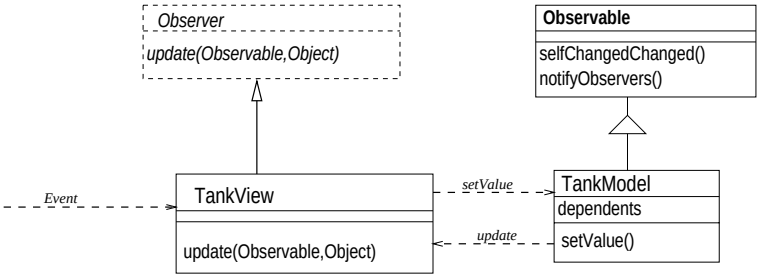


Abbildung 4.15: Ereignisbehandlung mit Java-Swing

4.4 Wiederverwendung Anwendungsrahmen, Entwurfsmuster, Komponenten

Beim Entwurf von Präsentation, Verhalten und Modularisierung einer Benutzungsschnittstelle müssen Konzepte des Software Engineering eingesetzt werden. Deren wesentliches Ziel ist das Wiederverwenden von bereits entwickelten Elementen. *Anwendungsrahmen* verwenden dazu das Erben von Funktionalität von ganzen Anwendungen; *Entwurfsmuster* vererben die Funktionalität von Teillösungen. Dagegen werden Anwendungen in der *Komponententechnologie* durch einen Einsteckmechanismus aus vorgefertigten Elementen zusammengesetzt.

In diesem Abschnitt werden die Konzepte Anwendungsrahmen, Entwurfsmuster und Komponenten eingeführt, soweit sie im Zusammenhang mit dem Entwurf von Benutzungsschnittstellen stehen. In Kapitel 5 wird dann eine dem Aspektansatz entsprechende Modularisierung von Benutzungsschnittstellen vorgestellt, die visuelle Spezifikationsmethoden für diese Konzepte ermöglicht; die Methoden selbst werden in Kapitel 6 vorgestellt.

4.4.1 Anwendungsrahmen (engl. *frameworks*)

Gut funktionierende Anwendungsrahmen sind wichtige Hilfsmittel bei der Entwicklung von Benutzungsschnittstellen. Der Begriff *Anwendungsrahmen* wird im Rahmen objektorientierter Softwareentwicklung verstanden als Entwurf für einen Anwendungsbereich; eine Menge kooperierender (auch abstrakter) Klassen, die eine implementierte Lösung darstellen. In diesem Sinne gibt es Anwendungsrahmen, mit denen der Anwendungsbereich „Programmierung von Benutzungsschnittstellen" abgedeckt wird, z. B. das Application Framework von VisualWorks [Par95], das Java Development Kit [Kra97], der JavaSwing-Anwendungsrahmen [Ost98] oder der Anwendungsrahmen HotDraw [Joh92].

Die Benutzung von Anwendungsrahmen erfolgt über den Erbungsmechanismus, d. h. die Framework-Klassen und ihre Beziehung werden geerbt, falls nötig modifiziert und dann zur Laufzeit ausgeprägt.

4.4.2 Entwurfsmuster (engl. *design pattern*)

Obwohl Entwurfsmuster im Software Engineering bereits seit knapp 10 Jahren verwendet werden, werden sie auf dem Gebiet der HCI erst seit einiger Zeit diskutiert und noch nicht verbreitet eingesetzt. Im Rahmen dieser Arbeit wurden von mir einige HCI-Entwurfsmuster gefunden und auf Workshops die grundlegenden Prinzipien ihrer Anwendung diskutiert. Aus diesem Grund werden Entwurfsmuster hier sehr ausführlich beschrieben.

Einführung

Der Begriff *„design pattern"* wurde von dem Architekten Christopher Alexander geprägt. Im Mittelpunkt seiner Forschung stehen die Empfindungen der Menschen, für die er Architektur entwirft. Er wollte herausfinden, was bestimmte Innenausstattungen, Räume und Gebäude besonders anziehend macht, warum einige Räume Menschen anziehen und sich in anderen Räumen Menschen nicht wohlfühlen. Dieses besondere Gefühl nennt er „Quality without a Name", also das „unbeschreibbare Wohlgefühl", das er, wenn es schon nicht beschreibbar

ist, umschreiben wollte. Die Umschreibung erfolgt als Muster in den Anordnungen von Gegenständen oder Architekturelementen. Umgekehrt sollen diese Muster auch als Entwurfshilfen dienen.

In seinen Büchern „A Pattern Language" [AIS77] und „The Timeless Way of Building" [Ale79] beschreibt Alexander diese Entwurfshilfen. Diese Vorgehensweise beruht darauf, in den Beschreibungen von Architektur-Aufgaben, -Problemen sowie einer bekannten und akzeptierten Lösung dazu Muster zu entdecken, die hervorrufen, daß Menschen sich wohlfühlen. Wenn solche Muster entdeckt werden, ist es möglich, Aufgaben und Probleme zu Klassen zusammenzufassen und allgemeine Lösungsschemata dafür aufzuschreiben. Eine Architekt/in, die eine Aufgabe zu lösen hat, informiert sich in einem Katalog solcher Aufgabenklassen, ob es schon eine Lösung für die Aufgabe gibt. Falls ja, nutzt sie das allgemeine Lösungsschema und paßt die Lösung der speziellen Aufgabe an.

Damit wird das unbeschreibbare Wohlgefühl durch ein Muster umschrieben, das einem allgemeinen Schema folgt:

- Beschreibung der Aufgabe bzw. des Problems
- Diskussion der Rahmenbedingungen und der von außen wirkenden Einflüsse auf das Problem
- Allgemeine Lösung des Problems
- Verwandte Aufgaben und Probleme

Ein Beispiel (siehe Abbildung 4.16) aus der Architektur von Wohnräumen beschreibt, wie man die Türen eines Raums so plaziert, daß Menschen, die sich in dem Raum aufhalten, möglichst wenig gestört werden (Muster 196 in [AIS77]).

Die Muster haben Alexander und seine Mitarbeiter/innen aus ihrer Erfahrung als praktizierende Architekt/innen gewonnen.

Kann eine Sammlung von Entwurfsmustern hierarchisch angeordnet werden, so daß die Elemente eines Entwurfsmusters durch ein oder mehrere weitere Entwurfsmuster ersetzt werden können, spricht man von einer *Entwurfsmustersprache* (engl. *pattern language*). Der Begriff kommt von der Vorstellung, daß, wie durch eine Grammatik, die Menge der gültigen Architekturen (mit dem unbeschreibbaren Wohlgefühl) durch sukzessive Anwendung der Entwurfsmuster gebildet werden, wie die Sätze einer (Programmier-)Sprache.

Eine nicht hierarchisch strukturierte Sammlung von Entwurfsmustern wird *Entwurfsmuster-Katalog* genannt.

Entwurfsmuster im Software Engineering

Diese Vorgehensweise wurde für den objektorientierten Software-Entwurf aufgegriffen, und zwar zunächst von Kent Beck und Ward Cunningham von Tektronix (siehe z. B. [Cun00]). Im Verständnis von Mustern beim Software-Entwurf geht es nicht darum, für Endbenutzer/innen ein unbeschreibbares Wohlgefühl herzustellen, sondern besonders elegante, effektive, wiederverwendbare und erweiterbare Programmentwürfe zu beschreiben.

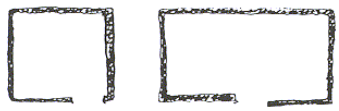
196 CORNER DOORS*

. . . this pattern helps you place doors exactly. Use it to help create the larger FLOW THROUGH ROOMS (131). You can use it too, to generate a SEQUENCE OF SITTING SPACES (142), by leaving small corners for sitting, uninterrupted by the doors; and you can use it to create TAPESTRY OF LIGHT AND DARK (135), since every door, if glazed and near a window, will create a natural pool of light which people gravitate toward.

❖ ❖ ❖

The success of a room depends to a great extent on the position of the doors. If the doors create a pattern of movement which destroys the places in the room, the room will never allow people to be comfortable.

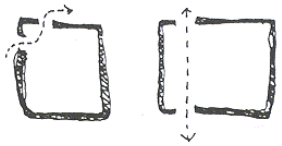
First there is the case of a room with a single door. In general, it is best if this door is in a corner. When it is in the middle of a wall, it almost always creates a pattern of movement which breaks the room in two, destroys the center, and leaves no single area which is large enough to use. The one common exception to this rule is the case of a room which is rather long and narrow. In this case it makes good sense to enter from the middle of one of the long sides, since this creates two areas, both roughly square, and therefore large enough to be useful. This kind of central door is especially useful when the room has two partly separate functions, which fall naturally into its two halves.



Rooms with one door.

Now, the case of a room with two or more doors: the individual doors should still be in the corners for the reasons given above. But we must now consider not only the position of the individual doors, but the relation between the doors. If possible, they should be placed more or less along the same side, so as to leave the rest of the room untouched by movement.

More generally, if we draw lines which connect the doors, then the spaces which are left uncut by these lines, should be large enough to be useful, and should have a strong positive shape—a triangular space left between paths of circulation will hardly ever be used.

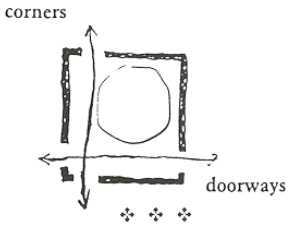


Rooms with more than one door.

Finally, note that this pattern does not apply to very large rooms. In a very large room, or in a room with a big table in the middle, the doors can be in the middle, and still create a special formal, spacious feeling. In fact, in this case, it may even be better to put them in the middle, just to create this feeling. But this only works when the room is large enough to benefit from it.

Therefore:

Except in very large rooms, a door only rarely makes sense in the middle of a wall. It does in an entrance room, for instance, because this room gets its character essentially from the door. But in most rooms, especially small ones, put the doors as near the corners of the room as possible. If the room has two doors, and people move through it, keep both doors at one end of the room.



❖ ❖ ❖

When a door marks a transition, as it does into a bedroom or a private place, for instance, make it as low as you dare—LOW DOORWAY (224); and thicken the entry way with closet space where it needs to be especially private—CLOSETS BETWEEN ROOMS (198). Later, when you make the door frame, make it integral with the wall, and decorate it freely—FRAMES AS THICKENED EDGES (225), ORNAMENT (249); except when rooms are very private, put windows in the door—SOLID DOORS WITH GLASS (237). . . .

Abbildung 4.16: Das Architektur-Entwurfsmuster „Ecktüren"
(Muster 196 in [AIS77])

Ein Software Engineering-Musterkatalog entstand aus den Diskussionen eines Workshops der OOPSLA'87. Erich Gamma erweiterte die Idee in seiner Dissertation ([Gam91]), und darauf aufbauend veröffentlichten Erich Gamma, Ralph Johnson, Richard Holm und John Vlissides einen Entwurfsmusterkatalog, der unter dem Titel *Design Patterns: Elements of Reusable Object-Oriented Software* [GHJV95] veröffentlicht wurde.

Hier ein Beispiel, das erklärt, wie beschrieben werden kann, daß sich ein Objekt in bestimmten ausgezeichneten Zuständen befindet (nach [ABW98] und [OS98]; die Beschreibung ist hier zusammengefaßt, sie umfaßt im Original ca. 10 Seiten):

Entwurfsmuster: State

Absicht:

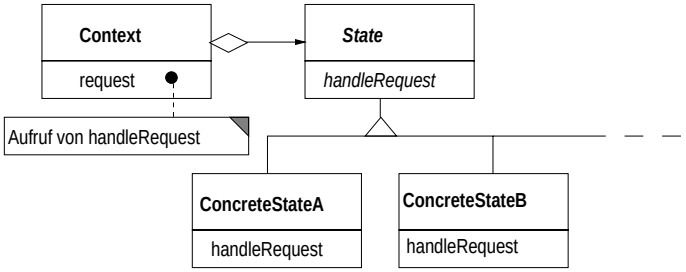
„Allow an object to alter its behaviour when its internal state changes. The object will appear to change its class."

Allgemeine Lösung

„The key to the State pattern lies in separating the state-dependent behaviour of the client into a separate set of State objects. The State object implement the part of the system's behaviour that is dependent on the current state of the client rather than the client itself."

Abbildung

Eine Abbildung zeigt die benötigten Objekte und ihre Beziehungen untereinander:



Beispiel

Als Beispiel kann man sich eine Diplomarbeitsthemen-Verwaltung vorstellen, bei der Diplomarbeitsthemen in den Zuständen „Kann bearbeitet werden (*unappointed*)" oder „Bearbeitung abgeschlossen (*appointed*)" sein kann. Je nach Bearbeitungszustand kann der Vorgang „Ausdrucken" etwas anderes bedeuten. Z. B. wird im Zustand „kann bearbeitet werden" ein Aushang für das Thema ausgedruckt, im Zustand „Bearbeitung abgeschlossen" die ganze Diplomarbeit. Solche Zustände können entweder in der Klasse selbst verschlüsselt werden - was bei vielen Zuständen zu großen Abfragen führt - oder als eigene Klassen, die der Hauptklasse zugeordnet sind, implementiert werden.

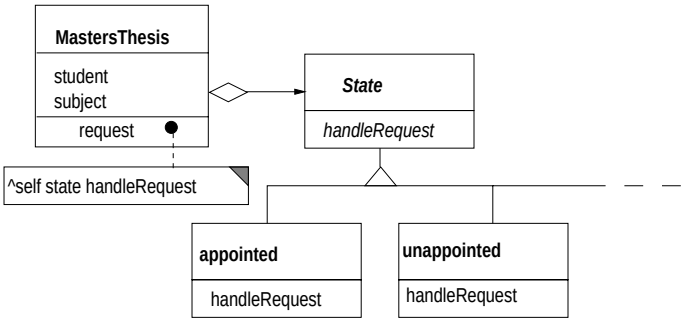


Abbildung 4.17: Beschreibung des State-Musters
(nach [GHJV95])

Die Beschreibung solcher Entwurfsmuster folgt einem ähnlichen Schema wie die Muster von Alexander.

Nach [GHJV95] besteht es im wesentlichen aus den vier Elementen[1]

1. Name des Musters
2. Problembeschreibung. Hier wird beschrieben, wann das Muster angewendet werden soll und auch der Kontext des Problems.
3. Die allgemeine Lösung des Problems beschreibt die Elemente, aus denen der Software-Entwurf zur Lösung des Problems besteht, ihre Beziehungen, Verantwortlichkeiten (engl. *responsibilities*) und die Art des Zusammenspiels.
4. Die Folgerungen aus dem Muster beschreiben die Ergebnisse und Vorteile des Musters.

Weiterhin wird eine graphische Skizze, meist mit Hilfe einer *Notation aus der Objektorientierten Analyse bzw. des Objektorientierten Entwurfs (*engl. *design)* (im Folgenden mit *OOA/D-Notation* abgekürzt) als Diagramm, zur Verdeutlichung genutzt. Das Arbeiten mit dieser Skizze kann durch ein Werkzeug, das die visuelle Programmierung benutzt, unterstützt werden.
Die 23 Entwurfsmuster in [GHJV95] sind für den Software-Entwurf (engl. *design*) aufgeschrieben, deshalb der Name Entwurfsmuster (engl. *design pattern*). Seit der Veröffentlichung des Buchs hat sich eine eigene Entwurfsmustergemeinde gebildet, die auch Konferenzen veranstaltet (z. B. die PLoP - Pattern Languages of Program Design - und EuroPLoP) und die Mailing-Listen und Veröffentlichungsreihen anbietet, so daß in vielen Anwendungsbereichen und auf allen Ebenen des Programmentwurfs Muster gefunden wurden. Entwurfsmuster, die im Anwendungsbereich „Entwicklung von Benutzungsschnittstellen" gefunden wurden, werden im nächsten Unterabschnitt vorgestellt.

Werkzeugunterstützung für den Einsatz von Entwurfsmustern im Software Engineering

Bereits gefundene Entwurfsmuster sollen beim Entwurf eingesetzt werden. Dadurch wird der Entwurf verständlich, entspricht in Teilen bestimmten Gütekriterien und kann effizient erfolgen. Ein Entwurfswerkzeug soll den Einsatz von Entwurfsmustern unterstützen. Auf der Entwurfsebene kann ein Entwurfswerkzeug, wie das in dieser Arbeit entwickelte Werkzeug COMBO, folgende Aufgaben übernehmen:

1. Ausprägen eines Entwurfsmusters, d. h. Bilden von Code-Schablonen für Klassen, die denen des Entwurfsmusters entsprechen.
2. Integrieren eines Entwurfsmusters in einen bestehenden Entwurf, d. h. bereits bestehende Klassen übernehmen die Rollen, die in dem Entwurfsmuster erforderlich sind. Rollen, die von keinen bestehenden Klassen übernommen werden sollen, können dann wie in 1. generiert werden.
3. Benutzen eines Entwurfsmusters, das bereits in einem Framework implementiert ist; beispielsweise wird das MVC-Muster in Smalltalk zur Entwicklung von Benutzungsschnittstellen verwendet (siehe auch Abschnitt 6.3.4).

[1] Welche Elemente notwendig sind, wird immer noch diskutiert und in der Literatur unterschiedlich angegeben, z. B. in [BMR+96, GHJV95, ABW98, CS95]

4. Finden eines Entwurfsmusters in einem bestehenden Entwurf.

5. Umgestalten eines bestehenden Entwurfs mit Hilfe von Entwurfsmustern.

Für das zweite Einsatzgebiet gibt es bereits einige Ansätze (z. B. [BFVY96], [Rat00]). In dieser Arbeit wurden Methoden für die ersten drei Einsatzgebiete entworfen und untersucht (siehe auch 6.6).

HCI-Entwurfsmuster

In diesem Abschnitt werden Muster vorgestellt, die speziell zur Entwicklung von Benutzungsschnittstellen dienen. Zunächst wird die historische Entwicklung der Forschung auf diesem Gebiet vorgestellt und die verschiedenen Versuche zur Klassifikation der Muster. Da es bis heute jedoch keine einheitliche Klassifikation gibt, teile ich die Muster zur leichteren Einordnung in folgende Kategorien ein:

- Gestaltung der Oberfläche (graphische Benutzungsschnittstellen-Elemente)
- Gestaltung der Interaktion mit der Endbenutzer/in
- Entwurf des Aufbaus der nicht graphischen Benutzungsschnittstellen-Elemente
- Entwurf von Verhalten der nicht graphischen Benutzungsschnittstellen-Elemente
- Entwurf der Anbindung an ein zugrundeliegendes Verarbeitungsprogramm

Nachdem der Entwurfsmusterkatalog von Gamma et al. erfolgreich bei der Entwicklung von Software eingesetzt wurde, wurden auch in anderen Bereichen Entwurfsmuster gesucht; insbesondere hat die HCI-Forschungsgemeinschaft die Ähnlichkeit der Gebiete Architektur und Entwurf graphischer Benutzungsschnittstellen-Elemente erkannt und sucht intensiv nach Entwurfsmustern. Entwurfsmuster gehen über die bekannten *Stilführer* (engl. *style guides*) und *Richtlinien* (engl. *guidelines*) hinaus, indem sie Beziehungen abstrakter beschreiben, als es die konkreten Vorschläge von Richtlinien tun, und sie beschreiben allgemeine Prinzipien, nicht Stile für konkrete Systeme.

Die verschiedenen, an der Benutzungsschnittstellen-Entwicklung beteiligten Disziplinen (Software Engineering, Interaktionsdesign, Layoutdesign,...) haben allerdings sehr unterschiedliche Sichten, Methoden und Vorgehensweisen. Während die Software-Entwickler/innen die oben genannten Qualitäten (Flexibilität, Erweiterbarkeit etc) in den Programmen erreichen wollen, stellen die HCI-Forscher/innen das unbeschreibbare Wohlgefühl der Endbenutzer/innen beim Benutzen der Benutzungsschnittstellen in den Mittelpunkt.

Daraus resultieren sehr unterschiedliche Entwurfsmusterkataloge und -sprachen, die u. U. schwer miteinander zu integrieren sind. In der HCI-Entwurfsmustergemeinde gibt es prinzipiell die Anhänger der „Alexander-Muster" und die Anhänger der „Gamma-Muster", was daran liegt, daß die Abgrenzung der genannten Disziplinen nicht sehr deutlich ist.

Ein Vergleich: Beim Bau von Gebäuden (siehe auch [Man97]) gibt es die Disziplinen Bauingenieurswesen und Architektur (einschließlich Innenarchitektur), die beide am Entstehen eines Gebäudes beteiligt sind und auch mit und an denselben Komponenten arbeiten. Demgemäß gibt es eine Mustersprache für Architektur. Um die Vorgehensweisen der Bauingenieure zu

beschreiben, sollte es eine weitere Mustersprache geben. Beide Sprachen würden die gleichen Gebäude beschreiben, denn trotz der Verflechtung enthalten die Entwurfsmustersprachen unterschiedliche Muster, wegen der verschiedenen Grundziele der beiden Disziplinen.

Eine einfache Übertragung der Architektur-Bauingenieur-Metapher würde Entwurfsmuster aufteilen in die Bereiche „Modellierung von Funktion der Benutzungsschnittstelle" bzw. „Gestaltung der Benutzungsoberfläche" (also des von der Endbenutzer/in erfahrbaren Teils der Benutzungsschnittstelle), ähnlich wie die Komponenten der Software-Architektur einer Benutzungsschnittstelle aufgeteilt sind (vergl. Seeheim-Modell und MVC, Abschnitt 4.2). Dabei wäre das Ziel der Funktionsmodellierung, Funktionalität sowie eine stabile Konstruktion zu garantieren. Ziel der Gestaltung der Benutzungsoberfläche wäre es, die Oberfläche so zu gestalten, daß die Endbenutzer/innen sich damit wohl fühlen. Diese Übertragung berücksichtigt verschiedene Punkte nicht: Die Verbindung zwischen Oberfläche und Anwendung ist nicht direkt einer der Kategorien zuzuordnen. Außerdem entwerfen die HCI-Entwickler/innen (Gestalter/innen der Oberfläche und Entwickler/innen des Oberflächenverhaltens) nicht nur die Anordnung von Elementen der Benutzungsoberfläche, sondern greifen in die Konstruktion ein. Noch gravierender ist, daß in der Praxis bei der Entwicklung vieler Benutzungsschnittstellen keine HCI-Expert/innen beteiligt sind, sondern daß die Software-Entwickler/innen oft auch die Oberfläche gestalten. Daher wird der Begriff HCI Entwurfsmuster von mir weiter gefaßt, wie im nächsten Abschnitt ausgeführt wird.

Definition und Strukturierung von HCI Entwurfsmustersprachen HCI Entwurfsmuster werden als eigenes Thema bis jetzt (Herbst 2000) hauptsächlich im Rahmen von Konferenz-Workshops diskutiert, da sich der Stand der Forschung noch nicht in allgemeingültigem Wissen verfestigt hat. Auch hat sich noch nicht eine allgemeingültige Terminologie durchgesetzt. Da im Rahmen dieser Arbeit der Ansatz, Wissen über den Entwurf von Benutzungsschnittstellen in Entwurfsmustern auszudrücken, sehr hilfreich ist, werden im folgenden die Berichte aus den Workshops zusammengefaßt dargestellt.

I. Der erste dieser Workshops fand während der **CHI'97** statt. Im Bericht wird nur ganz allgemein von Entwurfsmustern (Patterns) gesprochen, ohne sie speziell im Hinblick auf HCI zu benennen. Ziel ist „eine Entwurfsmustersprache für den Entwurf von Interaktion" (dieser Begriff wird nicht weiter erläutert, gemeint sind Entwurfsmuster der Kategorien Oberflächengestaltung und Gestaltung der Interaktion mit der Endbenutzer/in). Als Modell für die Entwurfsmuster wurde die Entwurfsmustersprache von Alexander gewählt.

II. Der nächste Workshop zu dem Thema fand auf einer der Entwurfsmusterkonferenzen, **ChiliPLoP'99** (*Hot Pattern Languages of Programming*) statt. Im Bericht über diesen Workshop wird die Terminologie „**Interaktionsmuster**" bzw. Entwurfsmuster für interaktive Anwendungen benutzt (ebenso Kategorien Oberflächengestaltung und Gestaltung der Interaktion mit der Endbenutzer/in).

Ein Ergebnis dieses Workshops ist die folgende Beschreibung:

> *Mit Hilfe von* ***Entwurfsmustersprachen für interaktive Anwendungen*** *(engl. interaction pattern language) entstehen solche Interaktionsentwürfe, bei denen die Vorstellung über das System dem konzeptuellen Modell der Endbenutzer/in über die zu erledigende Aufgabe (Task) sehr nahe kommt, um die Mensch-Maschine-Schnittstelle möglichst transparent zu halten (nach [Bor00a]).*

Außerdem wurde folgendes Klassifizierungsschema für „Interaktionsmuster" entwickelt:

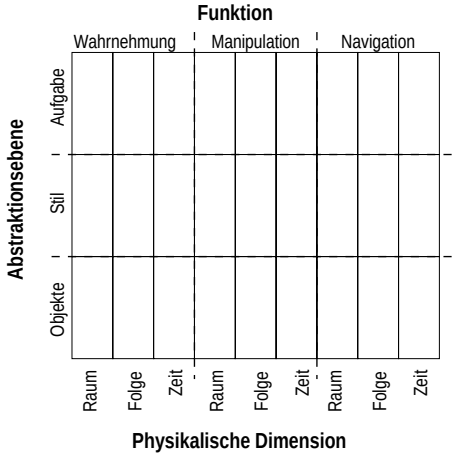


Abbildung 4.18: Taxonomie für „Interaktionsmuster" (Kategorien Oberflächengestaltung (und Gestaltung der Interaktion mit der Endbenutzer/in) nach *[Bor00a]*

III. Auf dem **INTERACT'99-Workshop „Usability Pattern Language: Creating a community"** wurde eine etwas andere Terminologie verwendet. Trotz des Titels des Workshops wird nun von *HCI-Entwurfsmustern* (engl. *HCI design pattern*) statt von Interaktionsmustern gesprochen, in Anlehnung an die Sprachregelung des CHI'97-Workshops. Über HCI-Entwurfsmustersprachen wird gesagt:

> *Das Ziel von HCI-Entwurfsmustersprachen ist, Personen, die beruflich mit dem HCI-Entwurf beschäftigt sind, erfolgreiche Entwurfslösungen zur Verfügung zu stellen. Damit wird allen, die mit dem Entwurf, der Entwicklung, der Evaluierung oder Benutzung von interaktiven Systemen zu tun haben, eine allgemein verständliche Sprache zur Verfügung gestellt (nach [BFG+99]).*

Diese Definition erweitert die Anwendung von HCI-Entwurfsmustern: Neben der Beschreibung von Entwurfslösungen für Benutzbarkeit von Benutzungsoberflächen (Kategorien Oberflächengestaltung und Gestaltung der Interaktion mit der Endbenutzer/in) ist nun auch die Beschreibung von Entwurfslösungen zur Erstellung von Benutzungsschnittstellen (Kategorien struktureller Aufbau und Verhalten nicht graphischer Schnittstellen-Elemente und Anbindung an ein Verarbeitungsprogramm) eingeführt.

In die Definition fallen auch alle Muster, die im Rahmen dieser Arbeit entwickelt wurden (siehe Abschnitt 4.4.2.2), d. h. Muster, aus denen eine Benutzungsschnittstelle konstruiert werden kann.

Muster, die eine ergonomisch sinnvolle Oberflächengestaltung erzeugen, werden in dieser Arbeit nicht weiter behandelt und sind Teil anderer Arbeiten (z. B. [Gri00a], oder das Einhalten von Guidelines z. B. [DH94]).

Ein Beispiel, das auch eine mögliche allgemeine Struktur von HCI-Patterns in Anlehnung an die Alexander-Muster zeigt, wird in Abbildung 4.19 dargestellt.

Workshop 3: HCI Pattern Language

The goals of an HCI pattern language

to share successful HCI design solutions among our colleagues

to provide a common language for HCI design to anyone involved in the design, development, evaluation or use of interactive systems

Example Pattern

Description @ Your Fingertips

Pattern title

Sensitizing example

You are putting interactive objects on a dynamic medium such as a screen and you want to provide various levels of context sensitive help supporting uninterruptable tasks.

Problem statement

Extensive explanations tend to clutter the interface but users may need such help. They do not want to leave the context of their current task, and experts may not want to see the help at all.

Existing examples

In the Mac OS, a small balloon of textual help appears when the user turns on this feature and moves the mouse over an object. In Windows Tool Tips the same happens if the mouse hovers over an object. In Netscape, the URL of a link is displayed in a fixed position at the bottom of the screen if the cursor is moved over it. In a voice mailbox, options are explained if the user waits for a while.

Formation of a general solution

therefore

provide a short description of the object either close to it or in a fixed position. Let users turn it on and off or only provide it on some explicit user action (e.g., hovering).

help

Schematic

Alternative representation:

```
On <start trigger>
   give <description>
   at <location>
On <end trigger>
   extinguish <description>
```

You can use three-state buttons to implement descriptions like this. Longer explanations can go into on-line help, possibly delivered via an intelligent agent, or in the manual.

Reference to constituent / related patterns

Contact

Richard Griffiths (r.n.griffiths@bton.ac.uk)

Lyn Pemberton (lp22@bton.ac.uk)

Jan Borchers (jan @tk.uni-linz.ac.at)

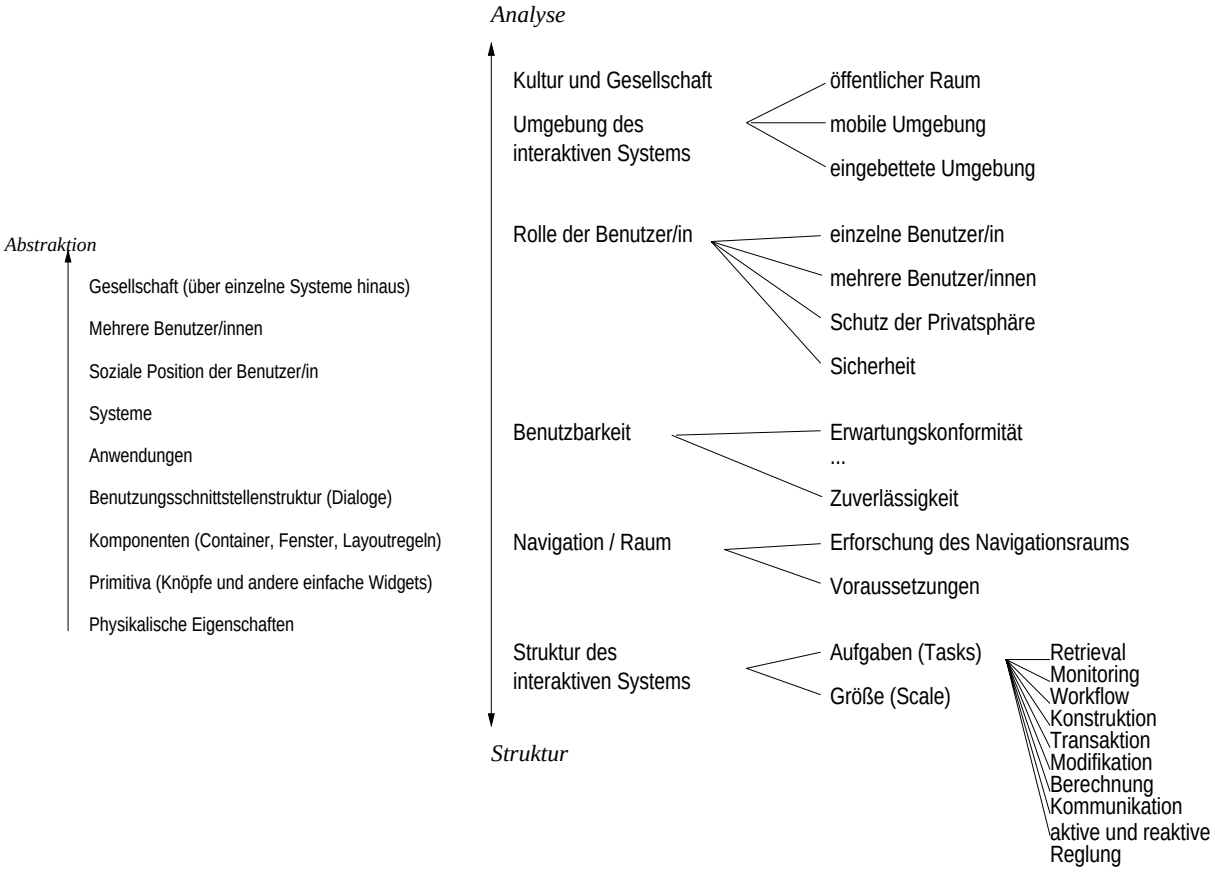
or visit the Web Site: http://www.it.bton.ac.uk/staff/rng/UPLworkshop99/

Abbildung 4.19: Ein HCI-Muster, das eine Mauszeigerinformation (engl. *tool tip*) beschreibt (nach [Gri00b], Kategorie Oberflächengestaltung und Gestaltung der Interaktion mit der Endbenutzer/in).

Eine „Standardform" für HCI-Entwurfsmusterbeschreibungen sollte meiner Meinung nach weitere Kriterien enthalten, z. B.

- *Bedingungen* (engl. *forces*), d. h. die Bedingungen und Kräfte, die den Einsatz des Musters erfordern (was in der obigen Beschreibung mit Kontext teilweise abgedeckt wird),
- *Implementierung*, damit sind verschiedene Implementierungsbeispiele für die Lösung gemeint,
- *bekannte Einsatzgebiete* (engl. *known uses*), d. h. Beispiele dafür, wo das Muster erfolgreich eingesetzt wird, und
- *Folgen* (engl. *consequences*), d. h. eine Beschreibung dessen, was passiert (positiv und negativ), wenn dieses Muster eingesetzt wird.

Außerdem wurden auf dem INTERACT'99-Workshop zwei weitere Klassifikationsschemata diskutiert. Diese beiden Klassifikationsschemata spiegeln die zwei Standpunkte der verschiedenen Disziplinen (Psychologie, Kognitionswissenschaften und Benutzbarkeitsingenieurswesen (die die Muster der ersten beiden Kategorien beschreiben) auf der einen Seite, Software Engineering und Computer Graphik auf der anderes Seite (die die Muster der letzten drei Kategorien beschreiben)) wider.



Konstruktionssicht
Klassifikation nach Abstraktion

Benutzer/innensicht
Klassifikation nach Struktur

Das im Beispiel beschriebene Muster „Description at your fingertips" würde also aus der Konstruktionssicht unter „Komponenten" eingeordnet werden, aus der Benutzer/innensicht unter „Navigation"

IV. Thema des Workshops auf der **CHI 2000** waren Sammlungen von HCI-Mustersprachen, wie sie zur Zeit entstehen:

- Benutzbarkeitsmuster [GPB00] (Kategorien Oberflächengestaltung und Gestaltung der Interaktion mit der Endbenutzer/in)
- einem konkreten Anwendungsgebiet zugehörige Muster [Bor00b] [Sie00] (Kategorien Oberflächengestaltung und Gestaltung der Interaktion mit der Endbenutzer/in)
- Muster, die Interaktionstechniken bekannter graphischer Benutzungsschnittstellen beschreiben [Wel00] (Kategorie Gestaltung der Interaktion mit der Endbenutzer/in)
- Muster, die die Konstruktion von Benutzungsschnittstellen beschreiben (Kategorien struktureller Aufbau und Verhalten nicht graphischer Schnittstellen-Elemente und Anbindung an ein Verarbeitungsprogramm)
- Muster, die auf einem der vorher stattfindenden Workshops diskutiert wurden [Fin00] (alle Kategorien)

Das Besondere an HCI-Mustern und Werkzeugunterstützung für ihren Einsatz
HCI-Muster beschreiben, ebenso wie die Architekturmuster, den Umgang mit „anfaßbaren" und sichtbaren Elementen. Dies führt dazu, daß auch der Einsatz von solchen Mustern sichtbar gemacht werden kann. Beispielsweise kann man sich für das „Description at your fingertips"-Muster (siehe Abbildung 4.19 auf Seite 85) die Integration in ein Werkzeug - z. B. einen Interface-Builder - vorstellen, bei dem solche Beschreibungsblasen auch anderen Elementen zugeordnet werden können.

Solche Überlegungen waren die Grundlage für die zwei von mir gefundenen Entwurfsmustersprachen, die im folgenden vorgestellt werden.

4.4.2.1 Eine Entwurfsmustersprache für Oberflächenverhalten (Kategorie Verhalten der nicht graphischen Benutzungsschnittstellen-Elemente)

Diese Sprache beschreibt, in welchem Kontext ein bestimmtes Verhalten von Elementen einer Benutzungsoberfläche eine gute Lösung ist [Sie00]. Im Rahmen dieser Arbeit wurde diese Sprache als Grundlage für die in Abschnitt 6.3.1 beschriebene Verhaltensbibliothek dokumentiert. Die Sprache gibt eine Einordnung der genannten Muster, sie legt *nicht* die Implementierung (z. B. durch Constraints oder Funktionen) fest.[2]

Eine Übersicht über die Sprache gibt Abbildung 4.20:

[2] Aus Platzgründen sind nicht alle bereits dokumentierten Muster aufgeführt.

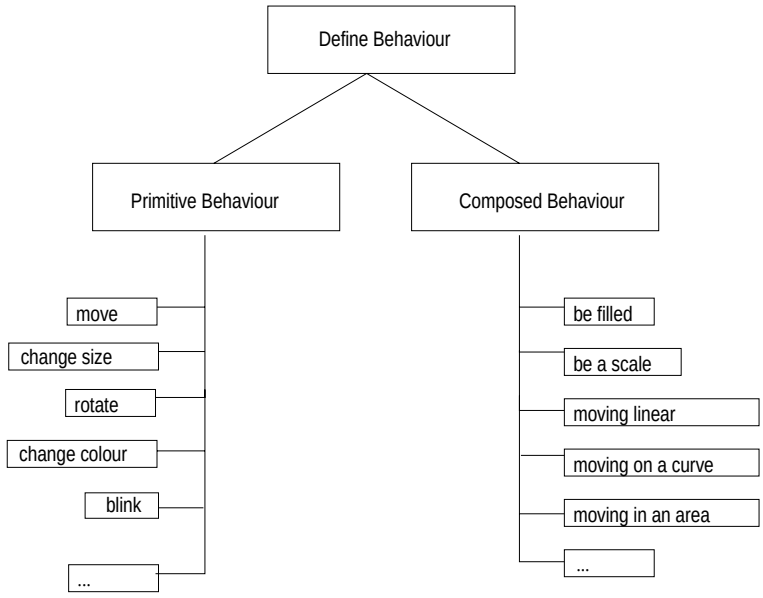
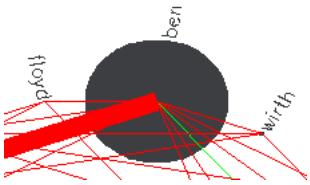


Abbildung 4.20: Eine Entwurfsmustersprache für Oberflächenverhalten

Im folgenden werden beispielhaft (sehr kurz) die Muster *„change size"*, *„be filled"* und *„moving in an area"* beschrieben.

Entwurfsmuster *Change size (Skalierung)*

Piktogramm



Kontext

Nachdem das Aussehen eines Benutzungsoberflächen-Elements festgelegt wurde und seine Attribute beschrieben wurden, muß nun spezifiziert werden, wie das Element sich verhalten soll, und durch welche Aktionen das Verhalten zur Größenänderung ausgelöst werden soll.

Problembeschreibung

Ein Oberflächenelement soll verschiedene Zustände, in Abhängigkeit von einem Zahlenwert, haben. Diese Zustände sollen der Endbenutzer/in verdeutlicht werden.

Beispiele

1. In einem lokalen Netz soll die Auslastung der Knoten und Verbindungen dargestellt werden. Dabei soll für die Endbenutzer/in intuitiv erkennbar werden, welche Knoten zum Betrachtungszeitpunkt besonders stark ausgelastet sind.

2. Ein anderes Beispiel ist die Darstellung der Füllhöhe eines Tankinhalts. Der Tankinhalt selbst soll durch ein Viereck dargestellt werden. Beim Anschauen des Vierecks soll für die Betrachter/in intuitiv deutlich werden, wie voll der Tank ist.

3. Oft sollen auch die Werte einer Tabelle visualisiert werden. Dabei sollen die verschiedenen Zahlenwerte unterschiedlich dargestellt werden.

Lösung

Visualisiere den Wert, der dargestellt werden soll, durch die Größe des Oberflächenelements. Dazu muß das Oberflächenelement seine Größenattribute in Abhängigkeit von dem Wert ändern. In den Beispielen ist die Lösung wie folgt angewendet:

1. Die Knoten werden als Kreise dargestellt, deren Größe sich in Abhängigkeit von ihrer Auslastung ändert.

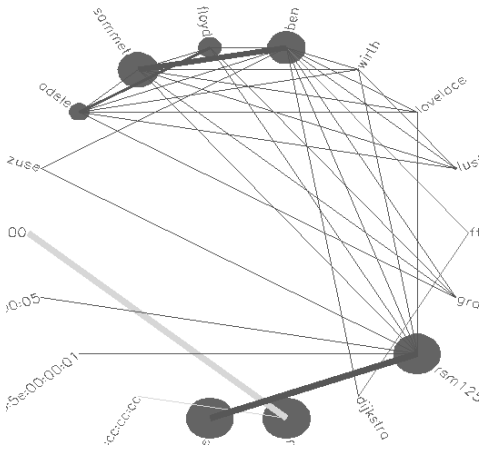


Abbildung 4.21: Darstellung der Kommunikation in einem Rechnernetz.
Die Größe der Knoten und die Breite der Verbindungen geben die Auslastung an.

2. Der Tankinhalt wird durch ein Viereck dargestellt, dessen Höhe sich in Abhängigkeit von dem tatsächlichen Tankinhalt ändert, siehe Abbildung 1.1. Die Linien geben die aktiven Verbindungen an.

3. Tabelleninhalte können durch die bekannten Balkendiagramme dargestellt werden.

Diagramm

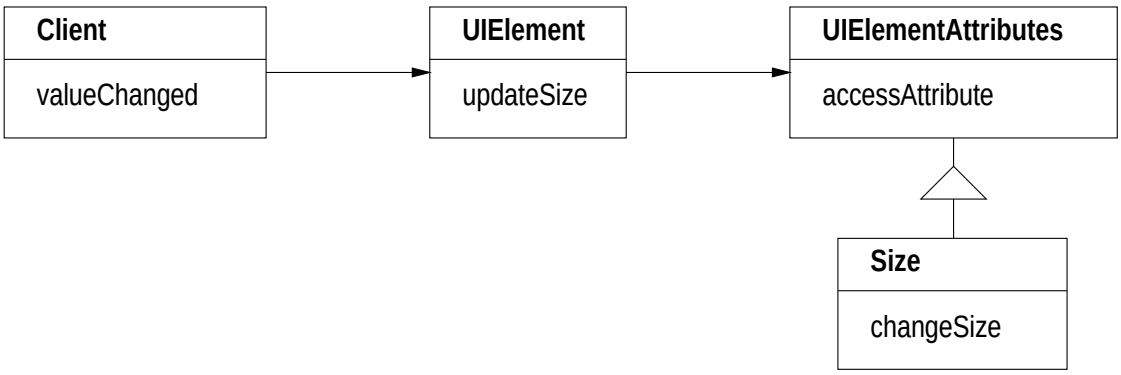
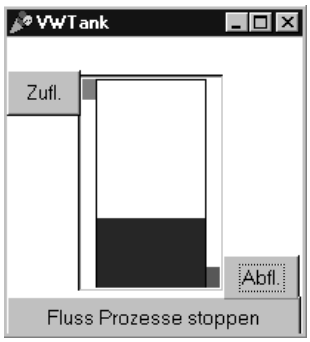


Abbildung 4.22: Klassendiagramm für das Entwurfsmuster *„Change Size"*

Entwurfsmuster *Be filled (Füllverhalten)*

PIKTOGRAMM



KONTEXT

Nachdem das Aussehen eines Benutzungsoberflächen-Elements festgelegt und seine Attribute beschrieben wurden, muß nun spezifiziert werden, wie das Element sich verhalten soll, und durch welche Aktionen das Füllverhalten ausgelöst werden soll.

PROBLEMBESCHREIBUNG

Ein Wert der Anwendung soll in Bezug auf eine Referenzgröße dargestellt werden. Der Wert drückt dabei ein Volumen oder eine Fläche eines Elements der Anwendung aus.

BEISPIEL

Das bereits eingeführte Tankbeispiel ist ein klassischer Anwendungsfall des Musters.

LÖSUNG

Stelle die Referenzgröße als Fläche dar, im Tankbeispiel als Rechteck. Teile die Fläche durch einen Strich in zwei Teile, wobei der eine Teil in der Regel andersfarbig ist. Die Lage des Strichs hängt von dem Wert der Anwendung ab. Diese Lösung wird meistens durch zwei übereinanderliegende Flächen dargestellt, wobei die eine Fläche - bis auf einen abgeschnittenen Teil - identisch mit der zweiten Fläche und in der Größe veränderbar ist. Zur Implementierung der veränderbaren Größe kann das *„Change Size"*-Muster verwendet werden.

DIAGRAMM

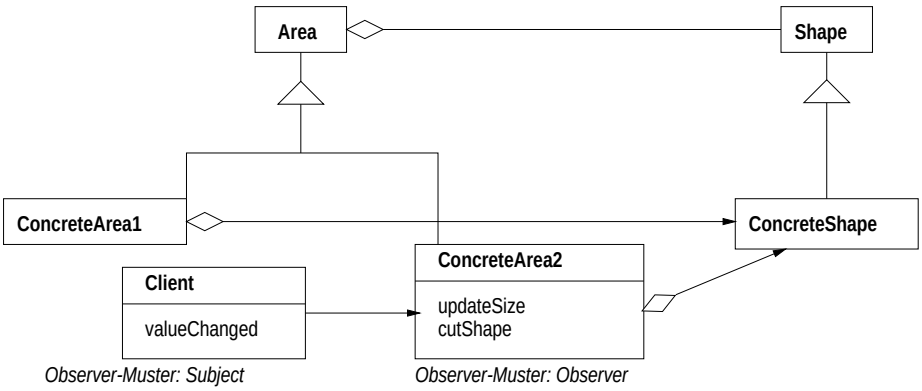
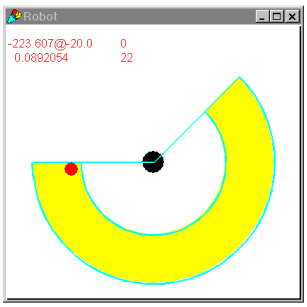


Abbildung 4.23: Diagramm für das Entwurfsmuster *„Be Filled"*

Entwurfsmuster *Move in an Area (Bewegung in einer Fläche)*

Piktogramm



Kontext

Nachdem das Aussehen eines Benutzungsoberflächen-Elements festgelegt wurde und seine Attribute beschrieben wurden, muß nun beschrieben werden, wie das Element sich verhalten soll, und durch welche Aktionen die Bewegungen ausgelöst werden soll.

Problembeschreibung

Die Position eines Elements der Anwendung soll auf der Benutzungsoberfläche im Verhältnis zu einem Bewegungsraum dargestellt werden.

Beispiel

Die Darstellung der Bewegungen des Greifers eines Roboters (siehe das Beispiel in Abschnitt 2.2.6 auf Seite 28) soll auf der Benutzungsoberfläche in Bezug zu dem möglichen Bewegungsraum des Roboters dargestellt werden.

Lösung

Der Bewegungsraum wird durch eine Fläche dargestellt und die Position des sich bewegenden Elements wird durch die Position eines Oberflächenelements in einer verhältnismäßig größeren Fläche ausgedrückt. Im Beispiel wird der Greifer durch einen roten Punkt und der Bewegungsraum durch einen Teil eines Kreises (genauer: durch ein Segment einer aus zwei Kreisen gebildeten Teilscheibe) dargestellt. Der Punkt als Darstellung des Greifers bewegt sich in Abhängigkeit von der realen Position des Greifers, bzw. wenn der Punkt bewegt wird, bewegt sich der Greifer mit.

Diagramm

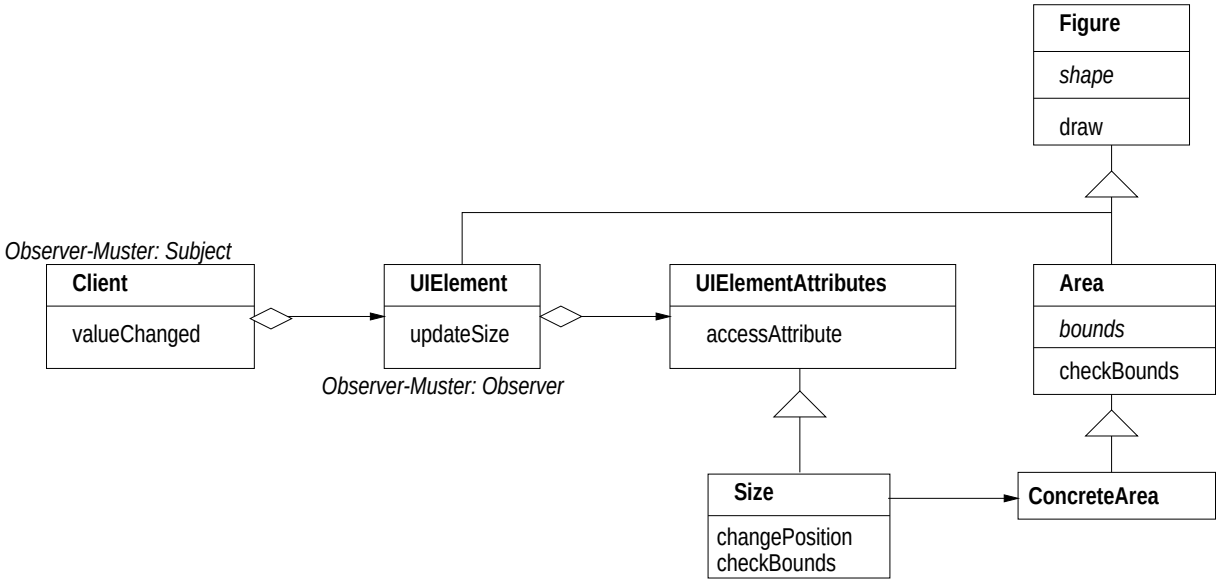


Abbildung 4.24: Diagramm für das Entwurfsmuster „*Move in an Area*"

Diskussion der vorgestellten Muster für Verhalten

Über den Wert solcher Muster kann man sich ausgiebig streiten[3]. Im Rahmen dieser Arbeit wurden die Muster vorgestellt, um eine alternative Werkzeugunterstützung des Entwurfs mit Mustern vorzustellen.

Eine Klassifikation der Muster in eines der oben vorgestellten Klassifikationsschemata ist möglich; sinnvollerweise würden die Muster nach Abstraktionsgrad geordnet, d. h. die komplexen Verhaltensmuster über den primitiven Verhaltensmustern. Da es in dieser Sprache nur Hierarchiestufen gibt, wie in der Übersicht zu sehen ist, ist diese Anordnung allerdings nicht besonders aussagekräftig.

4.4.2.2 Eine Entwurfsmustersprache für anwendungsspezifisches Layout (Kategorie Oberflächenverhalten)

Der Ausgangspunkt dieser Arbeit, nämlich die Konstruktion von technischen Benutzungsoberflächen liefert weitere gute Beispiele für anwendungsabhängige Entwurfsmuster. Wie bereits beschrieben, enthalten Benutzungsoberflächen für Automatisierungssysteme fast immer eine Kontroll- oder Menüleiste, einen graphischen Zustands- und Manipulationsbildschirm sowie eine Statusleiste. Daneben können auch hier die Richtlinien in Entwurfsmuster eingearbeitet werden, so daß die Kontrolleiste z. B. oben am Bildschirm angeordnet ist, der Status- und Manipulationsbildschirm in der Mitte und die Statuszeile am Fuß.
Ausgehend von dieser Grundstruktur können Entwurfsmustersprachen entwickelt werden, die die Anordnung (das Layout) und Funktion einzelner Elemente beschreibt.

Eine Übersicht über die Sprache gibt Abbildung 4.25.

[3]Ich möchte an dieser Stelle darauf hinweisen, daß ein Muster nicht eine Neuentwicklung darstellt, sondern gängige Praxis beim Entwurf eines Systems dokumentiert. Vielleicht haben Sie beim Durchlesen gedacht „aber das gibt es doch alles schon". Dann haben Sie diese Muster schon in Ihrer Entwicklungspraxis kennengelernt.

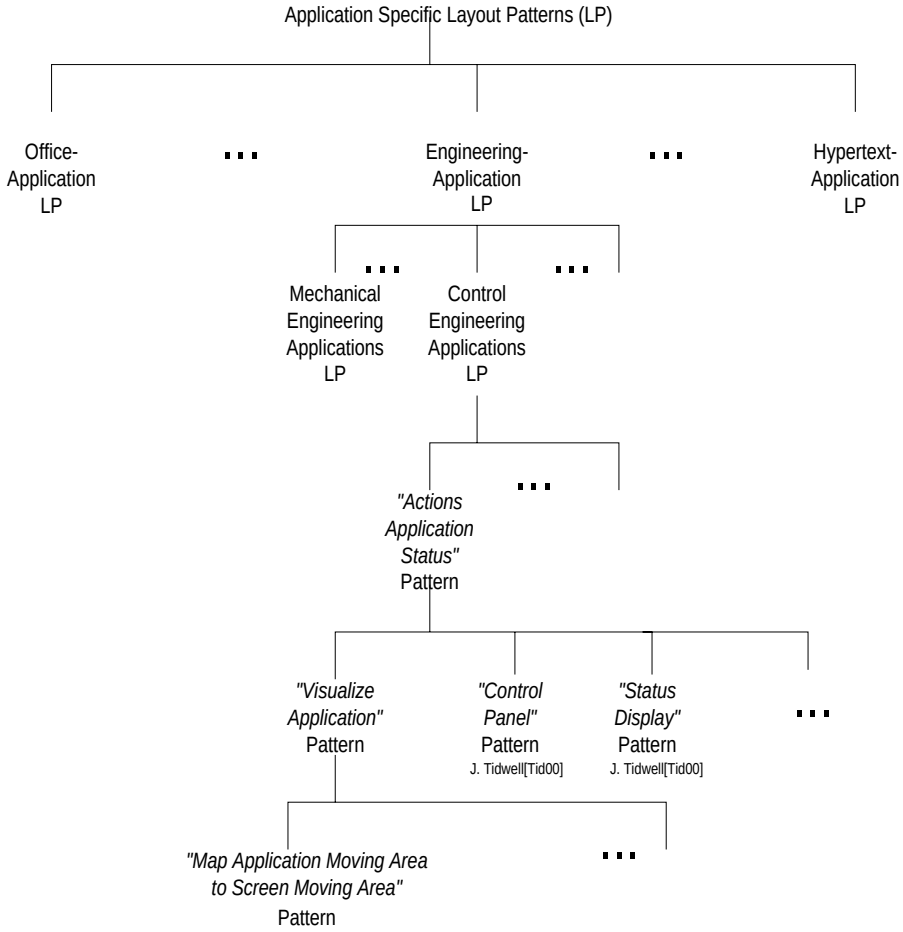
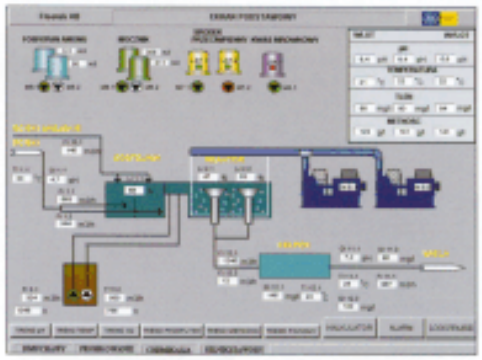


Abbildung 4.25: Eine Entwurfsmustersprache für das typische Layout von Automatisierungssystem-Benutzungsschnittstellen

Im folgenden werden beispielhaft die Muster *„Visualize Application"* und *„Visualize Application Movement Area" vorgestellt,* wieder sehr kurz.

Entwurfsmuster *Visualize Application*

Piktogramme



Kontext

Die Anordnung von Elementen in einem Automatisierungssystem erfolgt oft nach dem *„Actions, Application, Status"*-Muster. Es wird nun nach einer adäquaten Repräsentation der zu steuernden Anlage gesucht. Das Muster kann zusammen mit dem *„Control Panel"*- und *„Status Display"*-Muster verwendet werden [Tid00].

Problembeschreibung

Normalerweise werden Benutzungsschnittstellen, z. B. von Fabrikanlagen, von Fachkräften bedient, die die Struktur der Anlage kennen und die im Fehlerfall auch dafür verantwortlich sind, Maßnahmen zu ergreifen.

Daher sollte die Benutzungsoberfläche den Zustand der Anlage so darstellen, daß die Darstellung von Fehlerzuständen eine sofortige Zuordnung zu der Problemstelle möglich macht. Die Benutzungsschnittstelle sollte den Anlagenzustand so darstellen, daß die nötige Information gezeigt wird, ohne eine Bediener/in zu überlasten und von ihr zu viel Aufmerksamkeit zu fordern.

Beispiel

Eine Kläranlage soll von einer Person gesteuert und überwacht werden. [Fan99].

Lösung

Stelle die Struktur der Anlage als Graphikteil des Musters *„Actions, Application, Status"*. Benutze Abstraktionen, um die Struktur deutlich herauszuarbeiten, aber zeichne eine Grafik, die von der Struktur her der echten Anlage ähnelt, z. B. einen Tank durch eine vereinfachte Zeichnung eines Tanks. Zur Darstellung von Fehlerzuständen können das *„Normal operation - disruptive operation"*-Muster oder das *„Give a warning"*-Muster [Gri00a] verwendet werden.

Diagramm

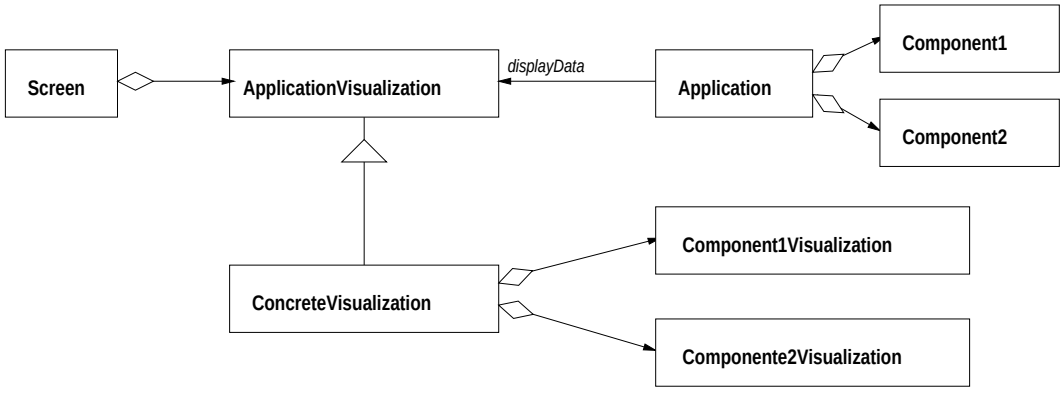
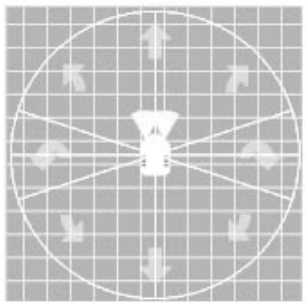


Abbildung 4.26: Diagramm für das Entwurfsmuster *„Visualize Application"*

Piktogramm



Entwurfsmuster *Visualize Application Movement Area*

Kontext

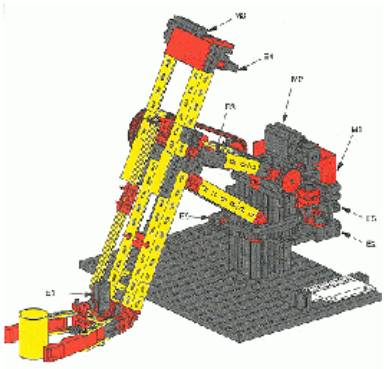
Das *„Actions, Application, Status"*-Muster beschreibt verschiedene Teile einer Benutzungsoberfläche und ihre räumliche Anordnung. Das *„Visualize Application"*-Muster kann verwendet werden, um den *„Application"*-Teil darzustellen. Eine Möglichkeit der Abbildung der Anwendungsstuktur ist das *„Visualize Application Movement Area"*-Muster.

Problembeschreibung

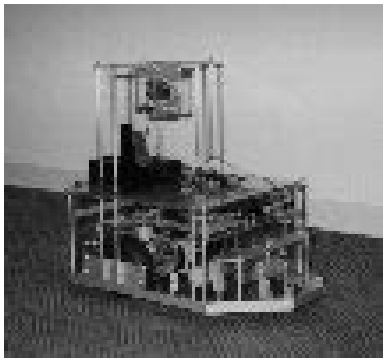
Ein Teil der Anwendung ist eine bewegliche Einheit, z. B. ein Roboter oder ein Fahrzeug. Nun muß eine Visualisierung gefunden werden, die die aktuelle Position der beweglichen Einheit wiedergibt und der Endbenutzer/in die Möglichkeit gibt, mit der beweglichen Einheit zu interagieren.

Beispiele

1. Ein Roboterarm mit einem Greifer ist auf einem Drehtisch montiert und kann den Greifer innerhalb eines bestimmten Raums bewegen. Der Roboter kann sich um 270 Grad drehen und kann die Kreismitte nicht erreichen. Somit ergibt die Projektion des Raums auf den Tisch als Bewegungsfläche einen Teil eines Kreises (genauer: durch ein Segment einer aus zwei Kreisen gebildeten Teilscheibe). Die Bediener/in der Benutzungsschnittstelle soll durch die Darstellung einen groben Überblick über die Greiferposition bekommen und soll mit der Darstellung den Greifer an eine andere Position bewegen können [SW00].



2. Ein Fahrzeug soll sich in einer rechteckigen Fläche bewegen können. Die Endbenutzer/in soll die aktuelle Position und Richtung, in der sich das Fahrzeug bewegt, aus der Darstellung ablesen und diese Richtung interaktiv ändern können, um das Fahrzeug zu steuern [CL00].

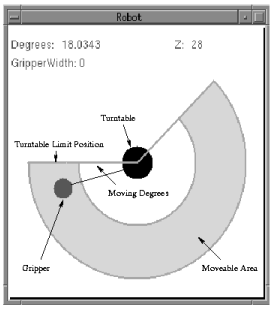


Lösung

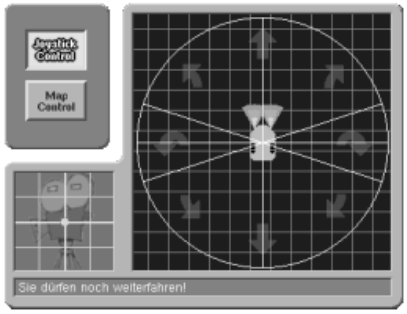
Zeichne die Bewegungsfläche als Fläche mit ähnlicher Form. Zeichne die bewegliche Einheit als Piktogramm innerhalb der Fläche. Die Endbenutzer/in kann die bewegliche Einheit interaktiv steuern, indem sie die Darstellung innerhalb der Fläche verschiebt.

Lösungen für die Beispiele sind:

zu 1. Zeichne die Bewegungsfläche des Roboters als Teil eines Kreises (genauer: durch ein Segment einer aus zwei Kreisen gebildeten Teilscheibe) und den Greifer als Punkt innerhalb der Fläche. Der Roboter kann gesteuert werden, indem der Punkt auf dieser Fläche verschoben wird.



zu 2. Zeichne die Bewegungsfläche des Fahrzeugs als Rechteck. Zeichne das Fahrzeug als Auto mit Scheinwerfern, wobei das Licht, das die Scheinwerfer ausstrahlen, die Richtung angeben, in der sich das Fahrzeug bewegt. Die Richtung des Fahrzeugs kann geändert werden, indem das Fahrzeug-Piktogramm gedreht wird.



Diagramm

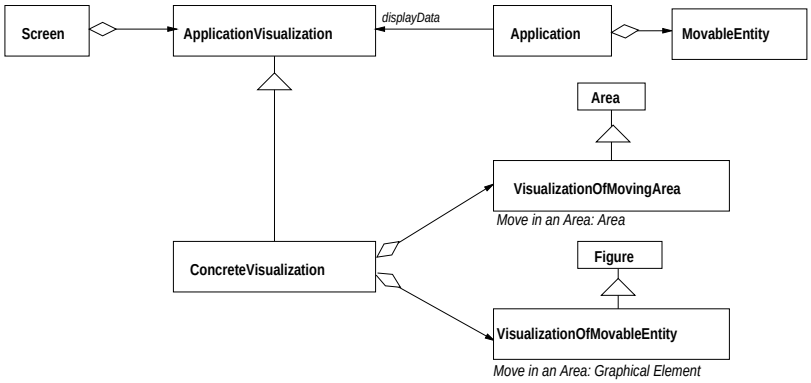


Abbildung 4.27: Diagramm für das Entwurfsmuster *„Visualize Application Movement Area"*

Formale versus informale Darstellung

In der HCI-Pattern Gemeinde gibt es eine rege Diskussion über den Aufbau von Musterbeschreibungen. Insbesondere bei der Beschreibung der Lösung durch Diagramme stellt sich die Frage, wie diese aussehen sollten. Die Schwierigkeit liegt darin, daß die Muster für Entwickler/innen, Designer/innen und Endbenutzer/innen (zum Verständnis der Benutzungsschnittstelle) verständlich sein sollen. In diesem Kapitel wurde die Lösung in Form eines Klassendiagramms dargestellt. Wie in Abschnitt 2.2.6 beschrieben, wäre z. B. eine Darstellung einer Fläche als Fläche anstatt einer Klasse „Fläche" intuitiv leichter zu verstehen, würde aber weniger konstruktive Hinweise für eine Software-Entwickler/in geben.

Eine Beschreibung könnte beide Formen enthalten und eventuell Abschnitte „für Entwickler/innen" und „für Endbenutzer/innen" einführen. Ein elektronischer Katalog könnte die Gesamtbeschreibung je nach Leser/in filtern.

Alternativ könnten die HCI-Muster auch in „Endbenutzer/innenmuster" und „Entwickler/innenmuster" unterschieden werden, was meines Erachtens keine gute Lösung ist, da diese Unterscheidung so nicht gegeben ist, z. B. wenn die Endbenutzer/innen auch Entwickler/innen sind. Die Lösung steht auch im Gegensatz zu der eigentlichen Absicht der Muster, Endbenutzer/innenbeteiligung bei der Entwicklung von Benutzungsschnittstellen zu fördern.

Werkzeugunterstützung für HCI-Entwurfsmuster

Ein umfassender Katalog von HCI-Entwurfsmustern aller Kategorien könnte eine wesentliche Unterstützung beim Entwurf von ergonomischen Benutzungsoberflächen bieten, bzw. die Einhaltung von Richtlinien garantieren. Da der in dieser Arbeit beschriebene Ansatz sich wenig mit Richtlinien befaßt, sondern vielmehr den software-konstruktiven Teil der Entwicklung von Benutzungsschnittstellen unterstützt, ist nur ein Teil der Muster interessant. Für die Integration in ein Werkzeug wie COMBO stellt sich insbesondere die Frage, wie HCI-Entwurfsmuster visualisiert werden können und welche Interaktionsmöglichkeiten für die Entwickler/in einer Benutzungsschnittstelle sinnvoll sind. Im folgenden wird dargestellt, welche HCI-Entwurfsmuster in ein solches Werkzeug integriert werden können und die mögliche Form der Integration:

- **Entwurfsmuster, die in einen Entwurfsmusterkatalog integriert werden können (alle Kategorien)**
  Ein rechnergestützter Entwurfsmusterkatalog, wie er in Abschnitt 6.6 für Software Engineering-Entwurfsmuster vorgestellt wird, könnte um HCI-Entwurfsmuster, die der geforderten Beschreibung genügen, erweitert werden. Dies trifft für Muster zu, die mit der von Gamma et al. verwendeten Schablone dargestellt und mit entsprechender Parametrisierung im Entwurf implementiert werden können. Ein Beispiel ist das ACEE Muster [Wu98], das im Zusammenhang mit der Entwicklung des Roboter-Beispiels aus Abschnitt 2.2.6 von Wu gefunden wurde. Es beschreibt den Kontrollfluß interaktiver technischer Anwendungen, die neben der Benutzungsschnittstelle eine oder mehrere Schnittstellen zu einem unabhängigen Aktor (z. B. dem Roboter) haben.

- **Entwurfsmuster, die eine spezielle, anwendungsabhängige Struktur der Benutzungsoberfläche beschreiben (Kategorien Oberflächengestaltung und Gestaltung der Interaktion mit der Endbenutzer/in)**
  Zu solchen Mustern gehören die auf S. 92 beschriebenen Anordungs- oder Layoutmuster.

- **Entwurfsmustersprachen, die eine besondere Visualisierung erlauben (alle Kategorien)**
  Einige Muster, auch im konstruktiven Software Engineering-Bereich, lassen sich leichter durch eine andere Visualisierung oder Interaktion erklären als durch Klassen- und Interaktionsdiagramme. Die Unterstützung solcher Muster kann dann mit Hilfe eines der anderen Aspekte, z. B. mit einer Methode zur Spezifikation des Layouts, erfolgen. Ein Beispiel sind die in Abschnitt 4.4.2.1 bereits vorgestellten Verhaltensmuster.

Vorteile von HCI-Mustern

Das Wissen von vielen Disziplinen (kognitive Psychologie, Ergonomie, HCI, Software Engineering, Anwendungsbereiche) kann in strukturierter Form abgelegt und wieder aufgerufen werden.

4.4.3 Komponenten

Ein wesentliches Merkmal von komponentenbasiertem Software-Entwurf ist das „Programmieren durch Zusammenstecken". Komponenten sind konzeptionell Programmteile, die unabhängig von anderen Programmteilen funktionieren.

Diese erste Beschreibung von Komponententechnologie wird in der Literatur unterschiedlich interpretiert, indem Komponenten entweder als programmiersprachliche Kapselung oder als Module aus Binärcode angesehen werden:

Komponentendefinitionen

Nach Booch et al. ist eine Komponente „ein physischer und ersetzbarer Teil eines Systems, das einer Schnittstellenbeschreibung entspricht und diese realisiert". In einem System finden sich verschiedene Arten von Komponenten, wie z. B. COM+ Komponenten, Java Beans oder auch Programmtextdateien als Produkte aus Entwurfsprozessen. Eine Komponente ist eine „physische Kapselung von logischen Elementen wie z. B. Klassen, (Java-)Interfaces und Kollaborationen" [BRJ97]. Szyperski definiert folgende Eigenschaften von Komponenten: Eine Komponente kann unabhängig ausgeliefert werden, (zur Laufzeit) durch Dritte zusammengesetzt werden und hat keinen persistenten Zustand [Szy99].

Programmierumgebungen, die Komponententechnologie unterstützen, definieren weitere Eigenschaften, die das Zusammenstecken der Komponenten ermöglichen. Beispielsweise müssen Java Beans auf Anfrage Informationen über sich selbst geben können, z. B. über ihren Namen und ihr Verhalten (was *introspection* und *reflection* genannt wird), außerdem muß ihr Zustand persistent gespeichert und wiederhergestellt werden können. Eine weitere Anforderung ist, daß ein Java Bean vollständig in Java (nach der Java Spezifikation) implementiert wird, was die Unabhängigkeit garantieren soll.

Stritzinger definiert eine Komponente abstrakter als „benanntes Objekt, das einen gekapselten Zustand speichert und dessen Abfrage und Manipulation durch öffentliche Aktionen mit exakt definierbarer Schnittstelle erlaubt" [Str97], wobei der Unterschied zu der Definition eines Objekts nicht mehr deutlich ist.

Nach jeder der Definitionen ist eine *Komponente* eine *Kapselung von Funktion,* die über eine fest definierte *Schnittstelle* aktiviert werden kann und die Funktionen anderer Komponenten aktivieren kann. Ein Schnittstellenteil, mit dem eine Funktion der Komponente aktiviert wird, wird in dieser Arbeit *Eingang* genannt, ein Schnittstellenteil, der andere Komponenten aktiviert, wird *Ausgang* genannt.

Zusammensetzen und Kommunikation von Komponenten

Komponenten existieren nicht nur unabhängig voneinander, sondern sie können auch zu Anwendungen zusammengesetzt werden. Zur Entwicklungszeit werden die Komponenten zusammengesteckt (was *Kopplung* genannt wird), und zur Laufzeit kommunizieren die Komponenten

miteinander, um das gewünschte Verhalten der Anwendung zu liefern (was *Kollaboration* genannt wird). Die Infrastruktur eines Systems, die das Zusammenstecken von Komponenten ermöglicht, wird *Komponentenbaukasten*, (engl. *component framework*), genannt.

Es werden verschiedene Kopplungs- und Kollaborationsarten unterschieden (nach [Str97]):

Kopplungsart	**Implementierung**	**Kommunikation über**
Direkte Kopplung	Komponente wird über Variable der Ausprägung referenziert	einfache Mitteilungen
Indirekte Kopplung	Zusammenstecken über Adapter	zusammengesetzte Mitteilungen (enthält Information über Zielkomponente)
Indirekte Kopplung: Protokollanpassung	Adapter kann Aufträge umformulieren, so daß andere Komponenten sie verstehen	einfache Mitteilungen
Indirekte Kopplung: temporärer Abruf	Adapter koppelt die Komponenten temporär statisch	zusammengesetzte Mitteilungen
Kopplung zur Laufzeit	erst durch die Art der Mitteilung bestimmt sich das Empfängerobjekt	zusammengesetzte Mitteilungen

Je nach Kopplungsart kann ein System verschiedene Arten der Kommunikation ermöglichen:

Kollaborationsart	**Kopplungsart**	**Bemerkung**
Statische Kollaboration	direkte Kopplung	Zur Laufzeit muß jeder Kommunikationskanal festgelegt sein, und die Komponenten müssen die Form der Mitteilungen kennen
Standardisierte Kollaboration	Protokollanpassung	Kommunikationskanal ist festgelegt, Form der Mitteilungen wird vom Adapter umgesetzt
Dynamische Kollaboration	Kopplung zur Laufzeit	Namen der Mitteilungen werden erst zur Laufzeit festgelegt

Tabelle 4.1: Kopplungsarten von Komponenten

In Entwicklungsumgebungen, die die Komponententechnologie unterstützen, wird in der Regel ein Ereignismechanismus, wie in Abschnitt 4.3.2 beschrieben, anstelle der Mitteilungen in der vorgestellten Klassifikation verwendet.

Das in dieser Arbeit verwendete Komponentenmodell

Komponenten können ganze Anwendungen sein, z. B. eine Tabellenkalkulation oder auch Teile davon. Für diese Arbeit sind Komponenten interessant, aus denen Benutzungsschnittstellen zusammengesetzt werden können. Beim Entwurf von Benutzungsschnittstellen werden Komponenten klassifiziert in *sichtbare Komponenten* und *nicht-sichtbare Komponenten*; sichtbare

Komponenten sind oder enthalten Benutzungsoberflächen-Elemente. Für das Komponentenmodell an sich ist diese Unterscheidung bedeutungslos, für den Entwurf visueller Methoden zum Zusammensetzen von Komponenten ist die Unterscheidung aber wichtig.

In Kapitel 6 werden solche Methoden vorgestellt. Sie beruhen auf dem **in dieser Arbeit benutzten Komponentenmodell**:

> Abstraktion von Funktion (als Zusammenspiel mehrerer Elemente) mit definierten Ein- und Ausgängen, die über Mitteilungen oder Ereignisse mit anderen Komponenten kommunizieren können.

4.5 Probleme bei der Integration von visuellen Ansätzen

Viele Werkzeuge zur Entwicklung von Benutzungsschnittstellen enthalten auch visuelle Ansätze zur Spezifikation.

Systeme zur visuellen Programmierung wurden in der Literatur vielfach ausführlich diskutiert, z. B. in [BB94, BGL96, Mye90, Shu88, IJK90, Gli90b, Gli90a, Pos96, Sch97a]. Eine Darstellung aller Systeme, mit denen auch Benutzungsschnittstellen generiert werden können, würde den Rahmen dieser Arbeit sprengen. Einige Werkzeuge werden im Zusammenhang mit den entsprechenden Methoden kurz vorgestellt.

Im Rahmen dieser Arbeit sind neben den Werkzeugen aus den Anwendungsbereichen auch die objektorientierten Programmierumgebungen von besonderer Bedeutung, da die meisten Systeme einschließlich der Benutzungsschnittstellen mit ihnen erstellt werden.

Einige dieser Werkzeuge bieten auch visuelle Methoden an. Es ist daher interessant zu erfahren,

- ob sie Unterstützung durch visuelle Ansätze für die Mechanismen bieten, die in Abschnitt 4.3 vorgestellt wurden,
- ob die in Abschnitt 4.4 vorgestellten Software Engineering Methoden visualisiert sind,
- ob sie eher auf Manipulation von *anwendungsspezifischen* Elementen oder auf Darstellungen von Mechanismen und Struktur basieren und
- wie diese visuellen Ansätze angenommen werden.

In diesem Abschnitt werden daher zunächst einige bekannte visuelle Ansätze in Programmierumgebungen (für Smalltalk und Java) vorgestellt. Weiterhin werden einige Schwierigkeiten verdeutlicht, die durch das Programmieren auf den in Abschnitt 4.3 vorgestellten unterschiedlichen Ebenen des Programmierens entstehen:

- Ebene des aufgabenorientierten visuellen Programmierens (Ebene 3)
- Ebene der visuellen Darstellung von Programmier- und Bibliotheksmechanismen und -strukturen (Ebene 2)
- Ebene der textuellen Darstellung von Programmier- und Bibliotheksmechanismen (Ebene 1)
- Programmiersprachenebene (Ebene 0)

Die visuelle Umsetzung

- des Verhaltens durch Mitteilungen (d. h. Methodenaufrufe) und
- des MVC-Musters bilden den Schwerpunkt der Betrachtung und
- der Ereignisbehandlung der Software-Komponenten.

Für die Spezifikation von Benutzungsschnittstellen wäre es gut, gerade auch im Hinblick auf Komponententechnologie, wenn Model-, View- und Controllerobjekte einfach „ineinandergesteckt" werden könnten. Leider ist das nicht so einfach, denn jede Implementierung hat andere zusätzliche Objekte, die berücksichtigt werden müssen: Adapter, Event Source- und Event Listener-Interfaces, Application Model, Wrapper ..... sorgen für konzeptuelle Modelle, die mit der einfachen Vorstellung von MVC wenig gemein haben.

Im Prinzip versuchen die Werkzeuge zu verhindern, daß sich Programmierende mit den Interna auseinandersetzen müssen, aber, praktisch gesehen, werden diese Interna benötigt, um Benutzungsschnittstellen zu erzeugen (sofern es sich nicht um Standard-Benutzungsschnittstellen handelt).

Zur Verdeutlichung wird auch hier ein Beispiel gegeben. Da die Entwicklungsumgebungen keine Tank-Widgets zur Modellierung des Tankbeispiels anbieten, wird als Beispiel die Verwaltung einer Liste von Namen verwendet. Listenwidgets, Eingabefelder und Knöpfe, sowie Listen-Klassen bieten alle Entwicklungsumgebungen an.

Die Akzeptanz der visuellen Ansätze wurde in einer Studie untersucht, die anschließend vorgestellt wird (Abschnitt 4.5.2).

4.5.1 Visuelle Ansätze in Entwicklungsumgebungen für Smalltalk und Java

Anhand der folgenden Beispiel-Entwicklungsumgebungen wird gezeigt, daß keine der bekannten Umgebungen dazu beiträgt, die genannten Programmierebenen konzeptuell für die Entwickler/in klar voneinander abzugrenzen.

Wichtiger ist jedoch, daß sie auch keine Hilfen geben, konzeptuell zwischen den verschiedenen Ebenen zu wechseln. Dieser Nachteil wird durch die daran anschließende Untersuchung bestätigt.

In Kapitel 5 wird das Aspektmodell vorgestellt, das die klare Abgrenzung und das Wechseln zwischen den Ebenen ermöglicht.

VisualWorks

VisualWorks enthält keine visuelle Programmiersprache für Benutzungsschnittstellen im engeren Sinne, sondern direktmanipulative Benutzungsschnittstellen-Entwurfswerkzeuge, die Programmtext für Benutzungsschnittstellen generieren.

Das Kernelement ist das `UIBuilder`-Werkzeug, ein Anordnungswerkzeug. Um eine Benutzungsschnittstelle vollständig zu erstellen, wird

- die Oberfläche mit dem Interface Builder erstellt (Ebene 3),
- die Anbindung erfolgt durch den sog. *Aspekt-Mechanismus (aspect mechanism)*,
- eine Ausprägung des Entwurfsmusters „Adapter" (textuelle Programmierung auf Ebene 1),
- die Anbindung zur Anwendung muß ausprogrammiert werden (Ebene 0), siehe Abbildung 4.28.

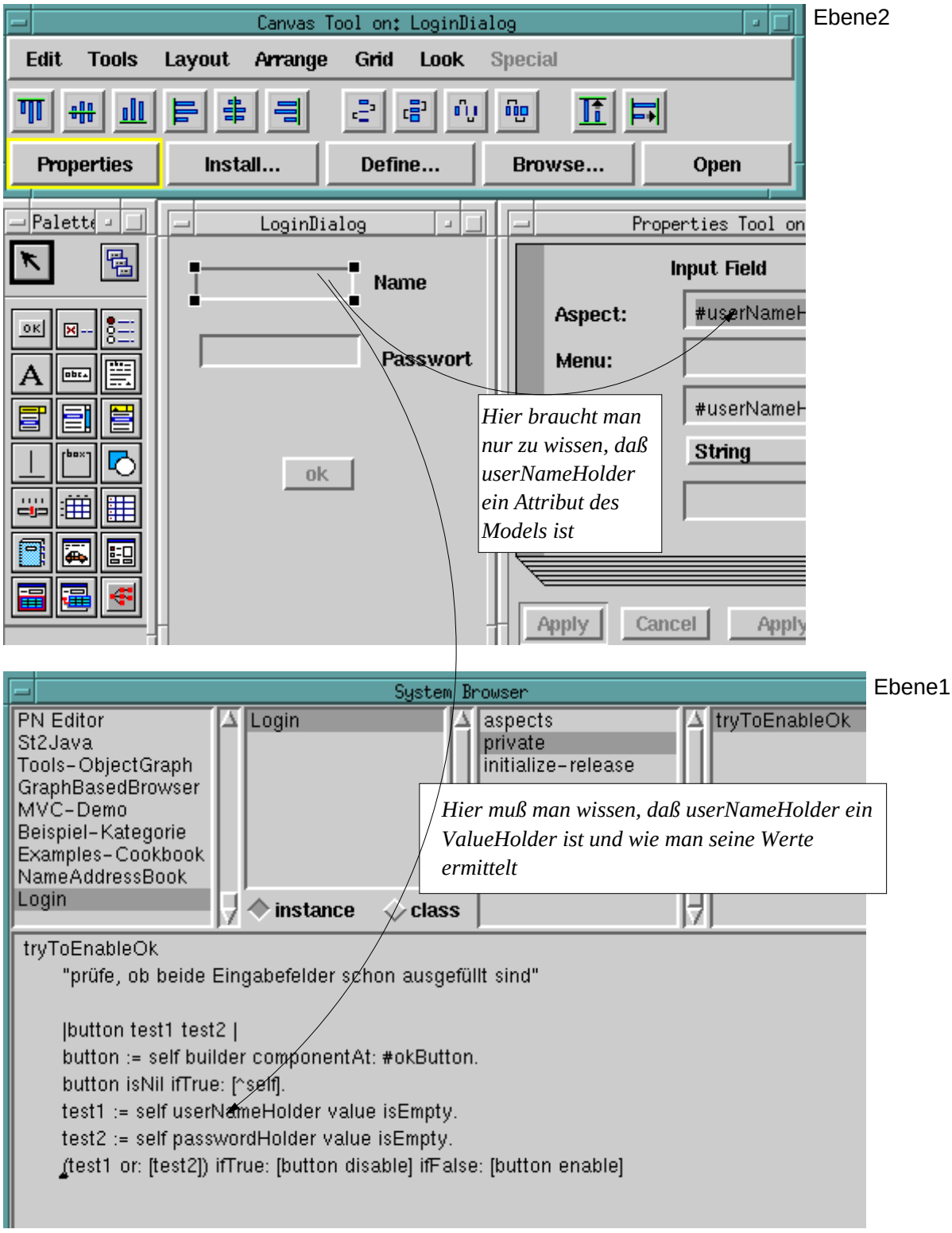


Abbildung 4.28: Programmierebenen bei Benutzung des Anordnungswerkzeugs von VisualWorks zur Programmierung einer Liste

PARTS für Java

In PARTS liegt der Schwerpunkt, wie schon aus dem Namen ersichtlich, auf der Entwicklung von Programmteilen, also im wesentlichen auf Komponenten, die *Parts* genannt werden. Statt dem in Abschnitt 4.4.3 beschriebenen Komponentenmodell benutzt PARTS das Java Beans-Modell, bei dem jede Komponente genau eine Klasse enthält und auf alle Methoden der Klasse zugegriffen werden kann.

Wie beschrieben gibt es auch hier sichtbare und nicht sichtbare Komponenten. Im Beispiel soll eine Liste mit Namen erstellt werden, die in einem Listenobjekt als Modell gespeichert sein soll. Der Name wird in ein Namensfeld eingegeben. Nach dem Editieren soll der Text in das Listen-Part eingefügt werden, und die gesamte Liste soll durch ein Listenwidget dargestellt werden, siehe Abbildung 4.29.

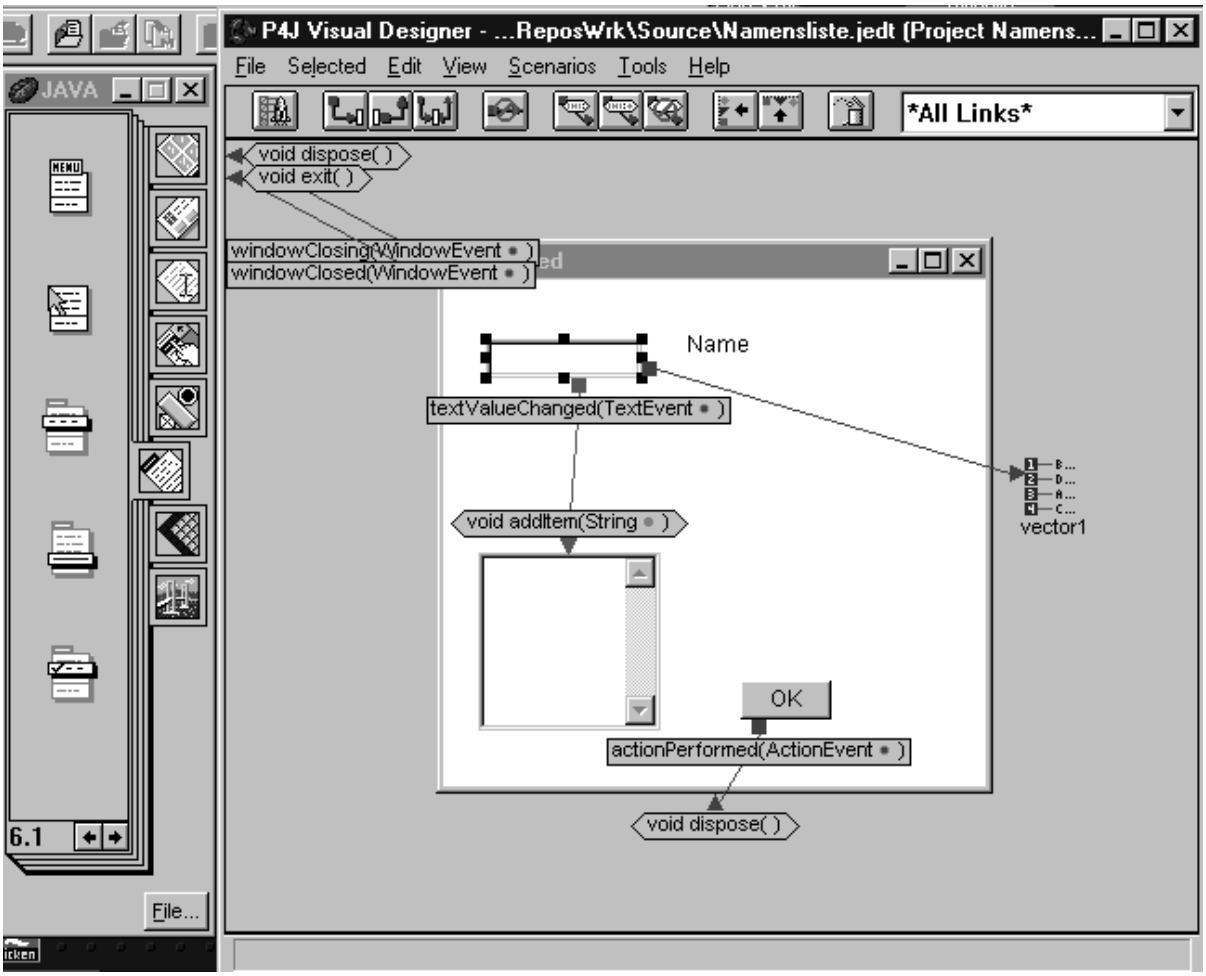


Abbildung 4.29: Programmierung einer Liste mit der PARTS-Technologie in PARTS für Java

Die Verknüpfung der Funktionalität zweier Parts wird durch Linien dargestellt. Im Beispiel ist zu sehen, daß von dem `Ok-Button` eine Linie zum Rahmen des Fensters geht. Die Verbindung wird dann aktiv, wenn das Ereignis `ActionPerformed(ActionEvent)` stattfindet, und es löst die Nachricht `dispose()` aus, d. h. bei Drücken des Knopfes wird das Fenster geschlossen. Die Linien beschreiben das Auftreten von und die Reaktion auf Ereignisse.

In einem Entwurfssystem, das sowohl die Ereignisse als auch die Mitteilungen/Methodenaufrufe sichtbar macht, müssen, um die Bedeutung einer solchen Linie zu definieren, mehrere grundlegende Entscheidungen getroffen werden: Handelt es sich um gerichtete Nachrichten? Entsprechen die Nachrichten den tatsächlichen Methoden der dahinter liegenden Objekte? Werden alle Methoden dargestellt, auch die geerbten? Woher weiß man, welche Methoden nun für den augenblicklichen Kontext wichtig sind? Oder handelt es sich um das Auslösen von Ereignissen? Welche Argumente bekommt das Ereignis?

Die Bedeutung von Verbindungen ist entscheidend für die Übereinstimmung mit dem konzeptuellen Modell der Entwickler/in. Wenn die Verbindung eine „abstrakte" Verbindung darstellt (zur Lösung einer Aufgabe der Benutzungsschnittstellen-Entwicklung), dann wird ein Modell auf der aufgabenorientierten Ebene (Ebene 3) angesprochen (dies ist z. B. bei JavaStudio, siehe Abschnitt 4.5.1, der Fall). Wenn die Verbindung einen konkreten Nachrichtenkanal bzw. den Aufruf einer Methode des programmiersprachlichen Objekts bedeutet, dann wird ein konzeptuelles Modell auf der Ebene der Mechanismen und Struktur (Ebene2) angesprochen.

Eine aufgabenorientierte Darstellung (auf Ebene 3) ist solange sinnvoll, wie weder neue Methoden geschrieben werden müssen noch Anbindungen an bestehende Anwendungen stattfinden müssen. Wenn die Entwickler/innen aufgabenorientiert (auf Ebene 3) programmieren müssen, um die Benutzungsschnittstelle vollständig zu definieren oder neue Parts zu definieren, sollte die Bedeutung so gewählt sein, daß ein Übergang zur Darstellung von Struktur und Mechanismen (Ebene 2) einfacher wird. Dies geschieht durch eine Sicht auf die textuelle Definition durch weitere graphische Sichten auf die gerade in der Entwicklung befindliche Anwendung.

Alleine das Fenster (der äußere Rahmen), in dem das Listenbeispiel dargestellt wird, hat ca. 50 Methoden. Da PARTS beim Ziehen einer Linie alle Methoden anzeigt (also ein textueller Blick auf die Struktur (Ebene 1) oder die Basismechanismen (Ebene 0)), aber ohne Erläuterung über die Funktionsweise der Methode, dauert es ziemlich lange, zu entscheiden, welche Methode die angemessene ist (außer die Entwickler/in weiß schon, welche sie braucht). Die Entscheidung erfolgt also auf Ebene 1 bzw. 0.

Als Modell wird ein Listenobjekt (z. B. eine `OrderedCollection`) verwendet, PARTS bietet z. B. einen *Vector-Part* an. Eine Durchsicht der Palette ergibt, daß hier eigentlich nur Ebene 3-Objekte vorhanden sind. Leider sind nicht alle Klassen der Klassenbibliothek auf der Palette vorhanden. Es kommt zu einer Vermischung von Ebene 3 und Ebene 2, die es schwer macht, mit dem Werkzeug umzugehen.

VisualAge

VisualAge für Smalltalk verwendet die Document View Architektur (siehe Abschnitt 4.2).

Zur Anbindung der graphischen Benutzungsschnittstellen-Elemente an die Anwendung benutzt VisualAge die PARTS-Technologie, d. h. es gibt eine Arbeitsfläche, auf der sichtbare (Widgets) und auf der Oberfläche nicht sichtbare Objekte, also Datenobjekte angeordnet werden können, siehe Abbildung 4.30 .

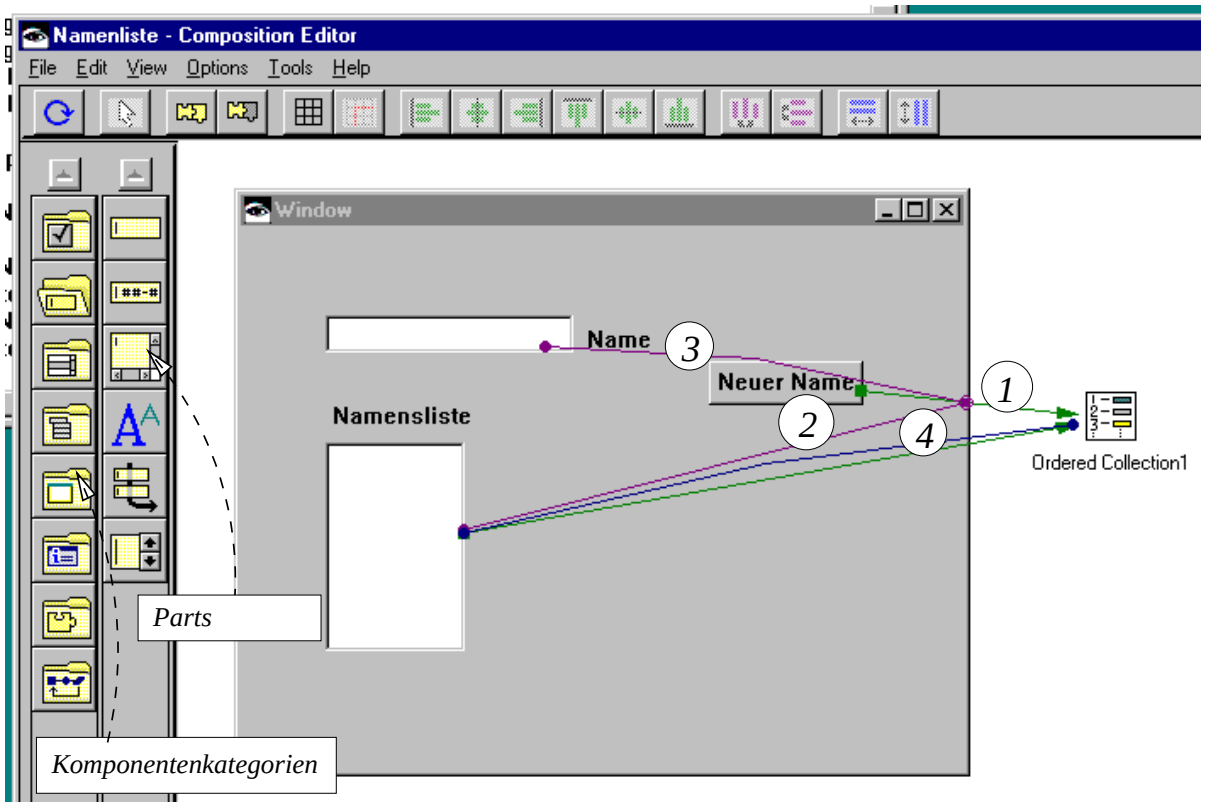


Abbildung 4.30: Benutzung der PARTS-Technologie zur Programmierung einer Liste im Werkzeug VisualAge. Die Zahlen bestimmen die Reihenfolge, in der die Linien gezogen werden.

Anstatt jedoch wie bei PARTS für Java nur eine Liste von Methoden als Reaktion auf ein Ereignis anzubieten, werden die Eigenschaften von Objekten aufgeteilt in Attribute und Methoden (genannt Aktionen), sowie mögliche Ereignisse aufgelistet. Zusätzlich gibt es noch die Möglichkeit, eigene Methoden als sogenannte Skripte zu schreiben. Daher können die Pfeile nun eine von vier Arten von Verbindungen ausdrücken: Attribute-to-Attribute, Event-to-Action, Event-to-Script und Attribute-to-Script. Darüberhinaus verbinden solche Pfeile nicht nur zwei Objekte, sondern gegebenenfalls mehrere Objekte. Beispielsweise wird folgender Sachverhalt in dem obigen Beispiel ausgedrückt: Beim Drücken des Knopfes „Neuer Name" wird ein neues Element in die Liste (der Klasse `OrderedCollection`) eingetragen (1). Der Eintrag ist eine Zeichenreihe, die der aktuelle Wert des Textfelds Name ist (3), d. h. der Parameter wird aus der Verbindung 3 hinzugefügt. Der Index, der die Stelle angibt, an der das Element eingefügt werden soll, bestimmt sich aus dem aktuellen Index des Listenelements (2). Verbindung (4) aktualisiert das aktuelle Listenelement. Das Beispiel ist ziemlich unrealistisch, und nur konstruiert, um zu zeigen, wie ein Modell benutzt werden kann.

Weder mit PARTS noch mit VisualAge kann das Tankbeispiel visuell programmiert werden

JavaStudio

JavaStudio ist ein ausschließlich (bis auf Markierungen) visuelles Programmierwerkzeug: Es gibt eine Anordnungs-Ebene (genannt Layout) und eine Datenflußebene (Design genannt), auf der die Komponenten miteinander verbunden werden können. Die Ein- und Ausgänge

sind dabei abstrakter als bei den PARTS-basierten Entwurfswerkzeugen, also nicht eine 1:1-Umsetzung der verfügbaren Methoden. Bei diesem Werkzeug wird nicht mehr auf der Ebene 0 programmiert, d. h. die Entwickler/in sieht nur noch die Komponenten und keine Klassen mehr.

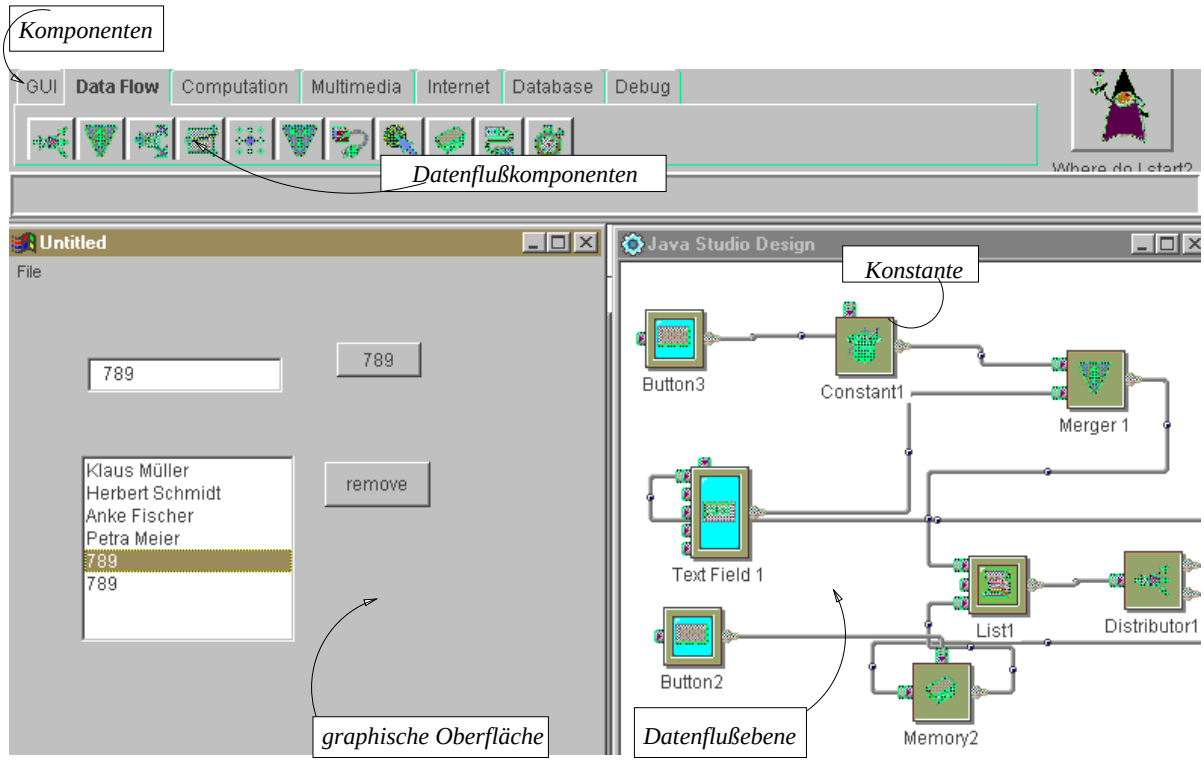


Abbildung 4.31: Programmieren einer Liste mit JavaStudio

Das Beispiel ist wieder eine Liste mit Namen. Da es bei der Auslieferung wenige nichtvisuelle Komponenten gibt, habe ich als „Modell" eine Konstante eingefügt, die bei Druck auf den Knopf 789 die Zahl 789 in die Liste schreibt. Die Datenflußdarstellung in leichter Anlehnung an Schaltpläne hat den Nachteil, daß man ständig solche Komponenten wie Verteiler einfügen muß, da z. B. ein Datum wie der momentan selektierte Index der Liste nur einmal aus der Liste herausgegeben wird, und zwar über denselben Port wie auch der Text der Liste. Im Prinzip sind nur Aktionen der Elemente zugänglich und nicht Attribute.

4.5.2 Untersuchung 3: Einsatz visueller Spezifikationsmethoden in visuellen Entwicklungsumgebungen für Smalltalk und Java

Wie die obigen Beispiele zeigen, ist visuelle Programmierung den Komponenten- und Ereignismodellen moderner Entwicklungsumgebungen für Java und Smalltalk angemessen. Sie wird aber nicht voll ausgenutzt, weil die visuelle Programmierung aufgabenorientierte (Ebene 3) und strukturorientierte (Ebene 2) Ebenen vermischt und die Entwickler/in zusätzlich die textuellen Basismechanismen (Ebene 0) und höherwertige Strukturen (Ebene 1) kennen muß.

These der Untersuchung

Für die verschiedenen Benutzer/innengruppen wurden folgende Thesen aufgestellt:

a) Anfänger/innen und **Endbenutzer/innen** verstehen die visuelle aufgabenorientierte Programmierung (auf Ebene 3) relativ leicht und verwenden sie auch, sind aber verwirrt, wenn es um die Anbindung an die Anwendung (Ebene 2) sowie Programmierung von Mechanismen auf Ebene 1 oder 0 geht, da diese im konzeptuellen Modell der Visualisierung auf der Strukturebene (Ebene 2) nicht enthalten sind.

b) Erfahrene Programmierer/innen haben die textuelle Programmierung auf den Ebenen 0 und 1 durchschaut und wollen nicht zusätzlich ein anderes konzeptuelles Modell bei der visuellen Programmierung (Ebenen 3 und 2) verwenden, da dieses zu umständlich wird.

Schwerpunkte der Studie

Ziel der Studie war es, herauszufinden, wie die visuellen Ansätze in ausgewählten Programmierumgebungen von den verschiedenen Entwickler/innengruppen akzeptiert werden. Dabei wurden sowohl die sog. Endbenutzer/innen (d. h. Personen, die wenig oder gar nicht programmieren können, teilweise z. B. Ingenieur/innen [SS96]) als auch Programmierer/innen einbezogen. Neben der Angabe, wie oft die Ansätze verwendet werden, wurden auch subjektive Bewertungen der Nützlichkeit sowie die Bewertung anhand der kognitiven Dimensionen ([GP96], siehe Abschnitt 2.2) erfragt.

Die Studie

Evaluationsmethode

In [Kit96] stellt Kitchenham neun Evaluationsmethoden von Software Engineering-Werkzeugen vor. Zwei Methoden sind zur Überprüfung solcher Umgebungen besonders geeignet: Ein kontrolliertes Experiment mit einer konkreten Aufgabe (wie das Experiment in Abschnitt 2.2.6) oder eine Umfrage.

Ein kontrolliertes Experiment wäre aus drei Gründen zu aufwendig gewesen:

- Um eine aussagekräftige Bewertung zu bekommen, hätten alle acht Werkzeuge von mindestens je drei Anfänger/innen bzw. Fortgeschrittenen evaluiert werden müssen; um statistisch relevante Ergebnisse zu bekommen, von noch mehr als einer Person. Darüberhinaus hätte die konkrete Aufgabe einen Programmieraufwand von ein- bis mehreren Tagen für die Versuchspersonen bedeutet. Ein solches Experiment ist unter den gegebenen Bedingungen, ohne Beobachtungshilfsmittel wie überwachbare Computer und Videokameras, nicht durchführbar.

- Die üblichen Versuchspersonen an einer Universität fallen fast alle in die Kategorie Anfänger/innen. Erfahrene Versuchspersonen hätten aus der Industrie angeworben werden müssen.

- Der Zeitaufwand ist für Versuchspersonen unangemessen hoch.

Daher wurde die Umfrage als Evaluationsmethode gewählt und durch einen Internet-Fragebogen realisiert. Nach [HGCM87] wird unterschieden in *Studien*, bei denen zuerst Informationen gesammelt werden, anhand derer eine Posititiv- oder Negativhypothese (z. B. über die Akzeptanz) aufgestellt wird, und *Tests*, bei denen zuerst die Hypothese aufgestellt wird, und dann Informationen gesammelt werden, die die Hypothese stützen oder verwerfen. Nach dieser Unterscheidung ist die Umfrage als Test einzuordnen.

Die Teilnehmer/innen konnten eine[4] beliebige „visuelle" Programmierumgebung für Smalltalk oder Java evaluieren, insgesamt waren es acht verschiedene Umgebungen mit visuellen Spezifikationsmethoden und zusätzlich zwei weitere, die jedoch auf C/C++ und PASCAL basierten.

Der Fragebogen bestand aus drei Teilen:

Teil I: Fragen zur Person, zur Umgebung und zu der Erfahrung mit dem Werkzeug

Teil II: Fragen über die visuellen Ansätze und zwar

- Subjektive Bewertung
- Wissen über Programmiermechanismen und Ereignisbehandlung
- Bewertung anhand der kognitiven Dimensionen

Teil III Fragen zur Bewertung des Bogens

Das Ausfüllen des Fragebogens hat zwischen einer und zwei Stunden gedauert; der volle Fragebogen ist in Anhang B abgedruckt.

Variablen

Folgende Variablen und ihre Messung wurden identifiziert, um die These zu verifizieren oder falsifizieren:

- **Benutzungsintensität der visuellen Ansätze** (subjektive Bewertung).
- **Zufriedenheit mit den visuellen Spezifikationsmethoden der Programmierumgebungen** (subjektive Bewertung auf einer Skala von 0 (schlecht) bis 6 (sehr gut)).
- **Benutzungsfreundlichkeit der einzelnen kognitiven Dimensionen der visuellen Ansätze** gemessen an der subjektiven Einschätzung der kognitiven Dimensionen (Bewertung jeweils auf einer Skala von 0 (schlecht) bis 10 (sehr gut)).
- Der **Grad der Übereinstimmung von konzeptuellem Modell der Benutzungsschnittstelle bei visueller bzw. textueller Repräsentation** wird eingeschätzt anhand des Verständnisses von Ereignis- und Programmiermodell (siehe Frage 6 in Anhang B). Wenn die Versuchspersonen nicht das Ereignismodell in der visuellen bzw. textuellen Repräsentation beschreiben konnten, oder diese Beschreibungen unterschiedlich ausfielen, wurde von einer Abweichung der konzeptuellen Modelle ausgegangen.

[4]Die Psychologie nennt dieses Verfahren „within-subject" im Gegensatz zu den „between-subject"-Verfahren, bei denen jede Person zu jedem Werkzeug eine Aussage machen muß.

Ergebnisse der Studie:

1. Allgemeine Daten

An der Umfrage haben 29 Personen gültig teilgenommen. Davon waren

Anfänger/innen	3
einigermaßen Geübte	6
Fortgeschrittene	20
Frauen	2
Männer	25
Geschlecht unbekannt	2

Tabelle 4.2: Beschreibung der Testpersonen

2. Welche Programmierumgebungen und Sprachen wurden verwendet?

Die benutzten visuellen Programmierumgebungen waren für die folgenden Sprachen konzipiert:

Smalltalk	11
Java	12
andere	6

Tabelle 4.3: Die in den Programmierumgebungen verwendeten Programmiersprachen

Die visuelle Sprache/Methode, die am häufigsten in den Werkzeugen verwendet wurde, war neben dem Anordnungsprinzip (z. B. die sog. interface builder) die PARTS-Technologie. Nur eine einzige Methode arbeitet voll visuell, nämlich JavaStudio.

Aus der Datenmenge, verteilt auf 10 verschiedene Werkzeuge, konnten keine statistisch relevanten Schlüsse gezogen werden, sondern nur Tendenzen sichtbar gemacht werden. Interessante Ergebnisse ergaben sich aus Widersprüchen bei der Beantwortung verschiedener Fragen, sowie aus den Kommentaren, die die Teilnehmer/innen den Fragebögen beifügten.

3. Benutzung der visuellen Ansätze

Zur Messung, in welchem Umfang die visuellen Ansätze genutzt wurden, dienten die Fragen nach der Erfahrung mit dem Werkzeug und nach dem Anteil der visuellen Programmierung am Gesamtaufwand der Benutzungsschnittstelle. Es ließen sich jedoch keine Zusammenhänge zwischen der Benutzungsintensität und der Art der Benutzungsschnittstelle oder der Erfahrung mit dem Werkzeug feststellen. Tendenziell werden Werkzeuge, die mehrere visuelle Ansätze anbieten (z. B. die die PARTS-Technologie benutzen), zu einem höheren Prozentsatz visuell programmiert als Werkzeuge, die beispielsweise nur ein Anordnungswerkzeug (z. B. einen interface builder) anbieten. Tabelle 4.4 stellt die Zusammenhänge dar:

Klassifikation	Erfahrung 0=wenig 10 = viel	Wieviel wurde visuell programmiert?	Werkzeug
graphische Anwendungen			
Baumeditor	5	60 %	Squeak
Appletts für Lehrzwecke	7	95 %	JavaWorkshop
Programmierwerkzeuge	7	20 %	PARTS für Smalltalk
Büroanwendungen			
Beispielprogramme nach Tutorial	4	100%	JavaStudio
Keine Angabe	5	-	JavaWorkshop
Beispielappletts	7	60 %	”
Keine Angabe	5	-	”
Analyse von Umfragedaten	7	40 %	”
Keine Angabe	5	20 %	VisualAge Java
Keine Angabe	7	95 %	”
Verwaltungs-Software	6	50 %	”
Dateneingabe	5	90 %	VisualAge Smalltalk
Marketing- und Vertriebssystem	6	80 %	”
Kundendaten-Verwaltung	7	90 %	”
Datenbank-Anwendungen für Marketing	7	50 %	”
Callcenter	6	20 %	”
Tabellenkalkulation	5	70 %	VisualWorks
Konzernbilanz	6	30 %	”
Textverarbeitung	3	100 %	VisualAge C++
Keine Angabe	7	-	Kawa
Dokumentverwaltung	6	10 %	VisualCafe
Keine Angabe	6	-	Sniff
Kostenstellenrechnung	7	20 %	VisualBasic
Datenbankoberfläche, verteilte Anwendungen	7	60 %	Delphi
Technische Benutzungsschnittstellen			
Robotersteuerungsoberfläche	3	30 %	Visual Works
Oberfläche für einen Fuzzy-Regler	6	80 %	”
Touch-Screen-Anwendung	-	100 %	Nowait
Kommunikationssoftware für das Internet	7	0 %	VisualAge Java, C++

Tabelle 4.4: Zusammenhang zwischen Werkzeugen und Anwendungen

4. Subjektive Bewertung der Nützlichkeit der visuellen Ansätze, Zufriedenheit

Die Bewertung der visuellen Sprachen/Methoden erfolgte zweistufig. Zunächst wurde geklärt, welche visuellen Sprachen/Methoden die Entwicklungsumgebung anbietet, und es wurde um eine subjektive Bewertung gebeten. Im zweiten Teil wurde nach einer Einschätzung anhand verschiedener Kriterien auf Grundlage der kognitiven Dimensionen gefragt. Die folgenden Tabellen geben Durchschnittsnoten über alle Teilnehmer/innen an.

In **Frage 4** wurde gefragt: Welche Möglichkeiten der „visuellen Programmierung" für Benutzungsoberflächen bietet die Entwicklungsumgebung? Wie hilfreich sind sie?

Auf einer Skala von 0 bis 10 erhielten die verschiedenen visuellen Präsentationen die folgenden Gesamtdurchschnittswerte:

Visuelle Präsentation	Bewertung
Interface Builder	3,75
Visuelle Darstellung der Klassenhierarchie	2,53
Komponentenpalette	2,92
Visuelle Darstellung der Verbindung zur Anwendung	2,58
Visuelle Programmiersprache	1,56
Andere	-

Tabelle 4.5: Durchschnitt der Bewertung der visuellen Spezifikationsmethoden

Die Durchschnittswerte für die einzelnen Werkzeuge sind in der folgenden Tabelle 4.6 dargestellt (ungültige Einträge wurden nicht aufgenommen):

		VA Java	VA Smalltalk	VA C++	PARTS Java	VisualWorks	VisualCafe	Java Workshop	JavaStudio	Nowait	Sniff	Delphi
a	Interface Builder	3,5	3	3	2	2	8	6	5	10	1	-
b	graph. Darst. der Klassenhierarchie	5	2	-	1	5	-	4,3	-	9	1	-
c	Komponenten-palette	6,6	2	-	2	5	9	5,5	5	10	-	1
d	graph. Darst. der Abhängigkeiten zw. Komponenten	7	5	-	3	5	2	3,5	5	8	1	1
e	visuelle Pro-grammiersprache	-	1	-	3	-	-	5	-	8	3	-
f	andere	-	-	-	-	-	-	-	-	-	-	-

Tabelle 4.6: Durchschnitt der Bewertung der visuellen Spezifikationsmethoden für die einzelnen Werkzeuge

Bei der Auswertung fielen die folgenden Punkte besonders auf:

- Unter Punkt d wird nach der graphischen Darstellung der Abhängigkeit zwischen Elementen der Benutzungsoberfläche bzw. der Anwendung gefragt. Diese Frage wurde nicht einheitlich beantwortet, denn teilweise wurden solche Abhängigkeiten unter dem Punkt e (visuelle Programmiersprache) bewertet, zum Teil unter diesem Punkt (Punkt d.) beantwortet (das ließ sich aus den Kommentaren schließen).
- Die PARTS-Technologie wird in den verschiedenen Entwicklungsumgebungen (PARTS für Java, VisualAge) unterschiedlich bewertet. Dafür sind zwei Erklärungen möglich: Entweder handelt es sich um individuelle Unterschiede in den kognitiven Vorlieben/Fähigkeiten der Entwickler/innen, die sich aufgrund der geringen Zahl der Antworten signifikant auswirken, oder die Unterstützung der PARTS-Technologie ist in den Umgebungen unterschiedlich gut. Welche Interpretation zutreffend ist, kann hier nicht entschieden werden.
- Die Bewertung der visuellen Sprachen/Methoden ist nicht enthusiastisch, d. h. die Noten liegen teilweise recht niedrig.

Die **Frage 7** beschäftigt sich mit der gleichen Variablen (Zufriedenheit), jedoch mehr unter dem Aspekt der Benutzungsfreundlichkeit der einzelnen visuellen Ansätze anhand der kognitiven Dimensionen. Im Gegensatz zur vorhergehenden Frage wird hier nicht global nach der Nützlichkeit der visuellen Sprachen/Methoden gefragt, sondern die einzelnen Eigenschaften der visuellen Sprache, sowie die Unterstützung einzelner Aspekte des Software-Entwurfs, bewertet. Hier sind die Ergebnisse differenzierter, aber deutlich positiver.

Eine mögliche Interpretation ist, daß die Unzufriedenheit nicht an der Qualität oder Handhabbarkeit der einzelnen Methoden liegt, sondern an einem anderen Faktor (nach der Hypothese der fehlenden Übereinstimmung der konzeptuellen Modelle auf den verschiedenen Ebenen).

Bei der Vorbereitung des Fragebogens konnte nicht davon ausgegangen werden, daß jede Teilnehmer/in den kognitiven Rahmen für visuelle Programmiersprachen nach Green und Petre kennt, daher wurden die Fragen so gestellt, daß die Dimension deutlich wurde. z. B. wurde folgendermaßen nach den mentalen Anforderungen und daran anschließend nach der Nützlichkeit für die einzelnen Aspekte der Benutzungsschnittstelle gefragt:

> *Frage 7 b: Angenommen, Sie würden eine Schulung für Programmieranfänger/innen, die aber Experten eines Anwendungsgebiets sind, zur Einführung in die Entwicklungsumgebung halten.*
> *1. Würde es genügen, den Schüler/innen den visuellen Teil zu erklären? (10 = ja, um Anwendungen zu schreiben, braucht man nur den visuellen Teil, 0 = nein, Anfänger/innen würden mit den visuellen Möglichkeiten die Prinzipien überhaupt nicht verstehen)*
> *2. Ich würde folgende Teile einer Anwendung mit Hilfe der visuellen Anteile erklären (bitte anklicken):*
> *Aussehen der Benutzungsoberfläche*
> *Verhalten der Benutzungsoberfläche*
> *Reaktion auf Ereignisse*
> *Verhalten des restlichen Programms*
> *Verbindung von Oberfläche und dahinterliegender Funktionalität*
> *Entwurf von Klassen*
> *Programmierung von Objektmethoden*

Die Interpretation der Antworten auf Frage 7b ist: Wenn man mit der visuellen Methode die Programmierung gut erklären kann, sind die Anforderungen niedrig.

kognitive Dimension	VA Java	VA Smalltalk	VA C++	PARTS Smalltalk	VisualWorks	Visual Cafe	Java Workshop	JavaStudio	Nowait	Sniff	Delphi
Ähnlichkeit der Abbildung	8,3	8,6	9	9	8,3	10	7,3	-	10	-	-
Konsistenz	6,25	7,6	7,5	-	9	10	7,75	-	9	-	-
Fehlerbehaftung										-	-
mentale Anford.	4,5	5	2	10	5	10	0,5		-	-	-
Abhängigkeiten sichtbar	7	7,3	8,5	8,5	1	0	3,3	-	5	-	-
Änderungsfreundlich	9	7,6	8,5	8,5	5	9	7,3	-	9	-	-
im Ganzen	7,6	6,6	5,5	5,5	7	10	8	-	5	-	-
Abschnitte	7,6	8	10	7	7	10	8,3	-	10	-	-
inkrementelle Entwicklung mgl.	9,6	8,3	9	8	8	10	9,3	-	10	-	-
Übersicht	7,6	6	4	4,5	2,6	10	6,6	-	6	-	-
Sekundäre Notation	7,3	7	8	8	3,6	0	3	-	-	-	-

Tabelle 4.7: Bewertung der visuellen Spezifikationsmethoden mithilfe der kognitiven Dimensionen (Bereich von 0 (schlecht) bis 10 (sehr gut))

5. Grad der Übereinstimmung von konzeptuellem Modell der Benutzungsschnittstelle bei visueller bzw. textueller Repräsentation.

Frage 6 beschäftigt sich mit dem Ereignismodell. Wenn die Versuchspersonen nicht die visuelle Darstellung des Ereignismodells beschreiben konnten, obwohl das Werkzeug eine solche bietet, wurde das interpretiert als Nichtübereinstimmung der konzeptuellen Modelle des Ereignismodells bei textueller und visueller Programmierung.

Folgende Interpretationen der Antworten sind denkbar:

Ereignismodell	visuelle Darstellung	Folgerung
nicht beschrieben	nicht beschrieben	keine Aussage möglich
beschrieben	nicht beschrieben trotz Vorhandensein	1. programmiert lieber textuell 2. Lücke im konzeptuellen Modell
beschrieben	nicht vorhanden	es gibt keine Alternative zur textuellen Programmierung
beschrieben	beschrieben	Teilnehmer/in hat das Ereignismodell und seine Visualisierung verstanden

Tabelle 4.8: Interpretationsmöglichkeiten für das Verständnis des Ereignismodells

Von den 29 Teilnehmer/innen zeigten 5 Personen eine solche direkte Nichtübereinstimmung, die auf eine Lücke im konzeptuellen Modell deutet; allerdings machten 12 Personen keine Angaben zu dieser Frage, was ebenfalls auf die Nichtübereinstimmung deutet. 2 Personen erkannten richtig, daß das Werkzeug keine visuelle Repräsentation für das Ereignismodell anbietet.

Ein Teilnehmer schreibt als Kommentar:

> [Das Event-Modell] *Ist mir etwas unklar. Das Event-Modell ist durch das AWT (Abstract Windowing Toolkit) in Java vorgegeben. In diesem Sinne benutzt jeder die vorgegebenen Klassen. Ich benutze nicht die in Visual Cafe vorgegebenen Mechanismen zur Event-Behandlung, da daraus eine nicht sehr übersichtliche, zentrale Event-Behandlung entsteht.*

Interpretation der Ergebnisse im Hinblick auf die These und einige Kommentare

Obwohl die Daten statistisch gesehen nicht für eine Bestätigung oder Verwerfung der These dieser Untersuchung (siehe Seite 107) verwendet werden können, lassen sich insbesondere der Widerspruch zwischen den Fragen 4 und 7 sowie die Antworten zum Ereignismodell in Richtung auf eine Bestätigung der These interpretieren. Auch einige Kommentare, die bei der Beantwortung des Fragebogens gegeben wurden, deuten darauf hin.[5]

- Wenn es eine aufgabenorientierte Verhaltensmodellierung gibt, wird diese gerne verwendet, egal ob visuell oder graphisch. Die folgenden drei Kommentare einer Entwickler/in über eine Smalltalk-Entwicklungsumgebung mit einer Skriptsprache zur Definition von graphischem Verhalten zeigt, daß es eine Lücke im Übergang von der graphischen Verhaltensprogrammierung zur Methodenprogrammierung gibt (Tippfehler sind Fehler in den Zitaten):

[5]Da die Teilnehmer/innen an der Umfrage zum Teil an der Entwicklung von Benutzungsschnittstellen-Entwicklungswerkzeugen, die auch diese Arbeit beeinflußt haben, beteiligt waren, z. B. bei GENIUS und PARTS (siehe Anhang A), halte ich die Aussagen für maßgebend.

„Es gibt in Squeak zum einen Standard-Dialoge, die man sehr leicht an seine Beduerfnisse anpassen kann. Zum anderen gibt es eine neue Richtung genannt Morphic, die Script-faehig ist und an Self und Fabric (?) angelehnt ist.

...

Die Fehler treten bei der Methoden-Programmierung auf und bleiben daher potentiell bestehen. Im Allgemeinen denke ich schon, dass man visuell weniger Fehler macht, aber auch weniger Freiheiten hat.

...

Am Anfang hätte ich mir mehr Grafik gewünscht, nun manchmal mehr Text."

Die folgende Aussage eines Entwicklers zeigt ebenfalls, daß der Übergang von visueller zu textueller Programmierung bei der Kombination Interface Builder/textuelle Programmierung nicht geglückt ist:

"*Das visuelle Programmieren ist ja ganz nett, aber letztendlich muß man doch alles von Hand programmieren, dann mache ich es doch gleich als Text*"

Der nächste Kommentar beschäftigt sich mit der Mächtigkeit der visuellen Werkzeuge:

„I was part of the team that developed PARTS (for Smalltalk) about 10 years ago. As is probably appearent, I think the tool never achieved it's potential. There was some very good and original ideas involved in PARTS, but the bottom line is that I would rather code a program in Smalltalk than produce with PARTS. I think one of PARTS' biggest problems is that it was both a GUI builder and a visual progamming tool, and it did an adequate job at both of these tasks but did neither extremely well (it was simply too much to try attack in a single tool)."

Aus einer solchen Aussage können folgende Konsequenzen für den Entwurf visueller Methoden gezogen werden:

1. Es muß klar abgegrenzt werden, welche Aspekte (im Sinne des Aspektansatzes) einer Benutzungsschnittstelle mit der visuellen Methode modelliert werden können.
2. Die visuelle Methode muß so mächtig sein, daß die Entwurfsaufgaben damit (im Rahmen des Aspekts) vollständig gelöst werden können.

Zu Punkt 1 gibt derselbe Teilnehmer folgenden Hinweis:

I think visual construction systems that target a narrow, constrained domain (e.g. GUIs) can in many ways be superior to text coded alternatives. I'm not sure visual construction can be used to solve problems of a more general nature.

Punkt 2 wird durch folgenden Kommentar bestätigt:

„Problem ist neben dem unglücklichen Eventmodell (siehe Frage 6) daß viel überflüssiger (hinsichtlich Änderungen sogar schädlicher) Code für das Setzen von Attributen generiert wird, wodurch man letztlich nur noch die Erzeugung der Objekte und deren Hierarchie als hilfreich empfindet. Das Layout ist nur gut unterstützt, wenn man ohne die in Java üblichen Layout-Manager arbeitet."

Teil b) der These, daß Entwicklungsumgebungen einfach aus praktischen Erwägungen von Fortgeschrittenen nicht genutzt werden, wird ebenfalls durch Kommentare erhärtet:

> *„Durch die Unmenge von Fenstern, Buttons, Icons, Dialogen, ... wird die Entwicklung eines Programms mit einer Entwicklungsumgebung (EU) zum Kampf gegen*
> *diese (Stichwort Aufgabenangemessenheit einer Benutzungsoberfläche nach DIN66234/8). In der Zeit, die man benötigt um das 1000-seitige Handbuch der EU zu lesen und das Ding auch nur halbwegs in den Griff zu bekommen, kann man das Programm dreimal von Hand schreiben. Außerdem fehlt einem, durch die Unzahl von automatisch generierten Klassen, der Überblick über das eigene Programm. Oft fällt es schwer den eigenen Code mit dem automatisch generierten zu verbinden. Außerdem sind die EUs wahre Ressourcenfresser (Speicher/HD/CPU-Power). Alles von Hand zu machen ist überdies wesentlich flexibler."*
>
> (Als Antwort auf die Frage, ob visuelle Ansätze verwendet werden) *„Nicht mehr! Ich fand es zu umständlich eine Fuktionalität zusammen zu klicken. Und der erzeugte Code war fehlerhaft!"*

Die Wichtigkeit von Visualisierung der Strukturen und Mechanismen einer Klassenbibliothek (d. h. Programmieren auf Ebene 2) kommt in folgendem Kommentar zum Ausdruck:

> Zunächst schreibt der Teilnehmer über das Programmieren auf Stufe 2:
>
> *„SNiFF ist bislang nur zum Source-Code handling brauchbar, macht da aber eine recht gute Figur. Die Visualisierung von Zusammenhaengen geht weit ueber blosse Klassenhierarchien hinaus, was mir gerade bei größeren Projekten wichtiger ist als die graphische Programmierung, die bei Tools wie VisualAge nicht wirklich zu skalieren scheint. Die Integration mit Java ist aber bei SNiFF+J recht gut gelungen und intuituv."*
>
> Weiterhin beschreibt der Teilnehmer die Wichtigkeit des Verständnisses von Stufe 1:
>
> *„Ohne den OO-Hintergrund ist zu befürchten, daß das Funktionsprinzip der GUIs unverstanden bleibt. Wichtige Designkriterien wie Wiederverwendbarkeit von Komponenten, z. B. durch Unterklassifizierung, würden wahrscheinlich nicht in Betracht gezogen."*

4.6 Zusammenfassung

In diesem Kapitel wurden verschiedene Architekturen, Funktionsweisen, und Entwurfshilfen von Benutzungsschnittstellen vorgestellt. Es wurde gezeigt, daß spezifische Architekturen und Funktionsweisen durch spezifische Vorgehensweisen unterstützt werden.

Aber nicht nur die Bauweise einer Benutzungsschnittstelle sollte Einfluß auf die Vorgehensweise haben, sondern vor allem die Vorstellung der Benutzer/in über die Bauweise. Auch

Besonderheiten von bestimmten Arten von Benutzungsschnittstellen sollten berücksichtigt werden. Diese Themen wurden in Kapitel 2 und 3 behandelt und im Zusammenhang mit Programmierumgebungen für Smalltalk und Java untersucht. Das Ergebnis der Untersuchung war, daß die subjektive Bewertung der visuellen Programmierung in solchen Programmierumgebungen schlecht war, obwohl die visuellen Spezifikationsmethoden gut bewertet wurden. Dieses Ergebnis wurde so interpretiert, daß eine Vermischung der aufgabenbezogenen (Ebene 3) und strukturorientierten (Ebene 2) visuellen Spezifikationsmethoden zu einem unklareren konzeptuellen Modell führt.

In den folgenden Kapiteln wird zunächst das Aspektmodell vorgestellt, das eine konzeptuelle Aufteilung von Benutzungsschnittstellen gemäß dem Aspektansatz beschreibt. Den einzelnen Aspekten dieses konzeptuellen Modells läßt sich zuordnen, welche Teile (des Codes) der Benutzungsschnittstelle ihn beschreiben. Aus der (landläufig üblichen) Vorstellung über jeden einzelnen Aspekt lassen sich neue visuelle Spezifikationsmethoden ableiten, die damit dem konzeptuellen Modell der Entwickler/in angepaßt sind.

In Kapitel 6 werden dann visuelle Spezifikationsmethoden für die einzelnen Aspekte beispielhaft vorgestellt, mit denen objektorientierte Benutzungsschnittstellen entworfen werden können.

Kapitel 5

Das Aspektmodell: Grundlage für die visuelle Spezifikation von Benutzungsschnittstellen

Aufbauend auf den Erkenntnissen der letzten drei Kapitel über

- Wissen und Vorstellung der Entwickler/innen,
- das Anwendungsgebiet Prozeßleitsysteme und
- die Konstruktion von Benutzungsschnittstellen

wird nun das *Aspektmodell* formuliert. Die Idee dabei ist, aus der Menge der in Kapitel 4 beschriebenen Programmierkonzepte diejenigen Grundbegriffe zu extrahieren, die Menschen benutzen, um Programmierkonzepte einzuordnen. Diese Grundbegriffe beschreiben ebenenübergreifend, plattform-, architektur- und implementierungsunabhängig Aspekte einer Benutzungsschnittstelle.

Weiter wird gezeigt, wie die verschiedenen, in Kapitel 4 genannten Programmierprinzipien und -konzepte sich in das Aspektmodell einfügen, und daß sich die Entwickler/innen durch das Aspektmodell leichter auf den Programmierebenen zurechtfinden können.

Das Aspektmodell dient als Grundlage für die Auswahl und Gruppierung der visuellen Spezifikationsmethoden in Kapitel 6. Diese visuellen Spezifikationsmethoden wurden mit dem Werkzeug COMBO prototypisch implementiert.

5.1 Motivation für das Aspektmodell

Wie bisher gezeigt, sind beim Entwurf einer Benutzungsschnittstelle verschiedene Elemente und Funktionsmechanismen auf den Elementen zu spezifizieren. Jede Klassenbibliothek und jedes Werkzeug bietet zur Erleichterung des Entwurfs unterschiedliche Kapselungen und Abstraktionen an. Aufbauend auf diesen Kapselungen und Abstraktionen konnten Spezifikationsmethoden und -werkzeuge entwickelt werden, die den Entwurf für die Entwickler/in weiter vereinfachen.

Einige dieser Kapselungen und Abstraktionen sind für verschiedene Entwicklungsumgebungen und Programmiersprachen gültig, z. B. Entwurfsmuster. Oft sind es jedoch Konzepte, die nur für eine Programmiersprache, ein Werkzeug oder eine Klassenbibliothek gültig sind: Z. B. sind die in Abschnitt 4.3 beschriebenen MVC-Mechanismen in Java und Smalltalk, ja sogar in unterschiedlichen Smalltalk-Entwicklungsumgebungen verschieden implementiert. Oder die Konzepte erlauben nur die Spezifikation auf einer Programmierebene und bieten keine Hilfe beim Übergang zu anderen Ebenen: Mit einem Interface Builder kann die Anordnung von (bereits fertigen) Widgets spezifiziert werden, es können aber keine *neuen* Widgets spezifiziert werden. Dafür muß die Entwickler/in das Programmieren „von Hand" erlernen.

Das führt zu verschiedenen Problemen:

1. Problem: Mangelnde Flexibilität

⊖ Die Methoden und Werkzeuge sind nicht flexibel einsetzbar. Insbesondere können nur bestimmte Arten von Benutzungsschnittstellen mit ihnen erzeugt werden (z. B. Datenbankabfrageformulare). Bei den in dieser Arbeit untersuchten technischen Benutzungsschnittstellen muß eine Entwickler/in oft wieder zur Basisprogrammierung zurückgehen, um die gewünschte Funktionalität zu erreichen. Dafür muß aber oft ein (für ungeübte Entwickler/innen) neues Programmiermodell gelernt werden, und es muß ein neues konzeptuelles Modell aufgebaut werden.

⊖ Die Methoden und Werkzeuge sind oft nicht integrierbar. D. h. die Entwickler/in muß sich auf eine bestimmte Methode oder ein bestimmtes Werkzeug festlegen, das unter Umständen für einige Aspekte nicht so komfortabel ist. Oft wird auch ein bestimmtes Werkzeug in einer Entwicklungsgruppe verwendet, und alle Entwickler/innen müssen, ungeachtet ihrer kognitiven Vorlieben und Fähigkeiten, die gleichen Methoden anwenden.

⇒ Lösung

Um diese Nachteile zu überwinden, ist das Aspektmodell so entworfen, daß der flexible Einsatz von Methoden ermöglicht wird: Jeder Aspekt kann von verschiedenen Methoden spezifiziert werden, je nach Wunsch und Fähigkeiten der Entwickler/in.

2. Problem: Eingeschränktes konzeptuelles Modell der Entwickler/innen

Ein weiteres Problem entsteht aus der Verwendung von Konzepten, die *genau einen* Mechanismus in *genau einer* Programmiersprache mithilfe *genau einer* Klassenbibliothek spezifizieren können:

⊖ Die Entwickler/innen sind gezwungen, die Benutzungsschnittstellen in genau diesen Konzepten zu beschreiben, ohne ein allgemeingültiges (d. h. übergreifendes) konzeptuelles Modell über Benutzungsschnittstellen aufzubauen. Solch ein übergreifenden Modell, bestehend aus allgemeinen Begriffen wie *Aussehen*, *Verhalten*, *Anbindung*, *Grundstruktur,* findet sich sowohl in den Erklärungen über Benutzungsschnittstellen (siehe Abschnitt 3.3.5) als auch in der Literatur zur Beschreibung der Konzepte - kann aber zur Spezifikation aus den genannten Gründen nicht eingesetzt werden.

⇒ **Lösung**

Ein Modell, das genau auf diesen Grundbegriffen aufbaut, ermöglicht den Aufbau eines übergreifenden konzeptuellen Modells. Bei der Konstruktion der Benutzungsschnittstellen kann die Entwickler/in Spezifikationsmethoden verwenden, die auf diesen Begriffen basieren. Durch eine Einordnung der sprach- oder bibliotheksabhängigen Mechanismen unter die übergreifenden Begriffe fällt auch der Übergang auf solche spezielleren Spezifikationsmethoden leichter.

5.1.1 „Aspekt" und „Modell"

Der Begriff *Aspekt* wird definiert als „Art der Betrachtung oder Beurteilung von etwas" [Mül85]. Ein Aspekt ist also eine bestimmte Teilsicht auf ein System. Solch eine Teilsicht kann willkürlich gewählt werden. Systeme werden durch die Betrachtung anhand von Aspekten vergleichbar, z. B. ermöglicht eine Betrachtung von Programmen unter dem Aspekt „Komplexität" bestimmte vergleichende Aussagen. Ebenso ermöglicht das Einbeziehen bestimmter Aspekte bei der Konstruktion von Systemen das Erreichen von Vollständigkeit (oder anderen Zielen).

Abstrakte Aspekte von Benutzungsschnittstellen sind z. B. die Benutzungsfreundlichkeit, die Komplexität oder die Geschwindigkeit. Aus dem Blickwinkel der Benutzungsschnittstellen-Konstruktion sind solche Aspekte wie Aussehen oder Verhalten interessant.

Der Begriff *Modell* wird in natur- und ingenieurwissenschaftlichen Gebieten als „– üblicherweise vereinfachte – Abstraktion der Wirklichkeit"[Han98] beschrieben. Modelle beschreiben abstrahiert einzelne oder mehrere Aspekte eines Systems. Modelle werden einerseits verwendet, um Systeme zu beschreiben, andererseits als Vorlage zur Konstruktion von Systemen. Die Vorgehensweise zur Konstruktion einer Benutzungsschnittstelle könnte daher sein, die verschiedenen Aspekte einer Benutzungsschnittstelle zu modellieren und dann nach dem Modell die Benutzungsschnittstelle herzustellen (d. h. zu implementieren) oder generieren zu lassen. Eine solche Vorgehensweise wird in der Literatur über Mensch-Maschine-Interaktion *modellbasiert* genannt. Solche Benutzungsschnittstellen werden ebenfalls modellbasiert genannt (siehe z. B. [Züh00, Pue99, HV96, FF93, SLN93, NFS+93, ABBK93, FGKK88]).

In den Ingenieursdisziplinen können viele unterschiedliche Modelle von der Realität oder Wirklichkeit eines Systems gemacht werden. Darüberhinaus werden in der Informatik verschiedene Wirklichkeiten erschaffen (von denen dann wieder verschiedene Modelle geschaffen werden können). Objekte oder Funktionen, aus denen eine Benutzungsschnittstelle besteht, existieren nur im Prozessor und Speicher eines Computers und sind somit nicht anfaßbar, sondern können nur in einer beliebigen Präsentation auf dem Bildschirm sichtbar gemacht werden. Wie oben bereits erwähnt, gibt es auch keine physikalischen Grundbausteine. Daher kann die gleiche Benutzungsschnittstelle aus unterschiedlichen Objekten, Software-Komponenten, Funktionen usw. bestehen, die von der Phantasie der Entwickler/in abhängen und davon, welche Programmiersprache, welche Plattform oder welche Programmiermodelle verwendet werden.

Die Modelle von Benutzungsschnittstellen beziehen sich also nicht auf „die" Wirklichkeit, sondern auf eine Vorstellung der Entwickler/in. Wie bereits beschrieben, führt die Berück-

sichtigung von Konstruktionsdetails, die sich aus der Wahl der Plattform, Sprache, Bibliothek etc. ergeben, dazu, daß die Entwickler/in sich sehr um Details der Implementierung kümmern muß. Andererseits ist ein zu abstraktes Modell wenig sinnvoll, wenn der Bezug zur Implementierung verlorengeht: Das Seeheim-Modell, das in Abschnitt 4.2 vorgestellt wurde, beschreibt die Trennung von Präsentation und Dialogsteuerung sehr gut. Um damit Benutzungsschnittstellen spezifizieren zu können, muß es aber verfeinert werden, was z. B. mit Hilfe von Zustandsautomaten für die Dialogsteuerung geschieht.

Da in dieser Arbeit die Konstruktion von Benutzungsschnittstellen im Mittelpunkt steht, wurde ein Modell entwickelt, das durch die Auswahl der Aspekte *konstruktiv für Benutzungsschnittstellen* ist und das *konzeptuelle Modell der Entwickler/in berücksichtigt.*

Konstruktiv heißt, daß aus der Spezifikation der einzelnen Aspekte eine lauffähige Benutzungsschnittstelle generiert werden kann.

Zur Berücksichtigung des konzeptuellen Modells erfolgte die Auswahl der Aspekte aus den Erkenntnissen der Studie über den Entwurf von Automatisierungssystemen (siehe Abschnitt 3.3.5) zusammen mit den Erkenntnissen aus den Kapiteln 4 und 2.1.1.

Die Trennung in die Aspekte erlaubt auch eine Modularisierung dessen, was zur Konstruktion der Benutzungsschnittstelle spezifiziert werden muß. Dadurch, daß diese Modularisierung anhand von gleichartigen Teilsichten erfolgt, kann ein konsistentes konzeptuelles Modell aufgebaut werden.

5.2 Die Aspekte

Es ergeben sich vier wesentliche Aspekte, die eine Benutzungsschnittstelle beschreiben:

- Die *Anordnung der Elemente* (Widgets) auf der Oberfläche und ihren *graphischen Attribute*,
- die *Struktur* als Bauplan der
  - Benutzungsschnittstelle, also den Oberflächenelemente und allen sonstigen Elemente, aus denen die Benutzungsschnittstelle aufgebaut ist, sowie der
  - Anwendung, die durch die Benutzungsschnittstelle ergänzt wird,
- das *Verhalten* des Systems,
- die *Verbindung* oder Abhängigkeiten zwischen der Benutzungsschnittstelle und der Anwendung.

In den folgenden Abschnitten werden zunächst die Aspekte genauer beschrieben und ihr Vorhandensein in den einzelnen Teilen einer Benutzungsschnittstelle gezeigt.

Als Beispiel dient die schon erwähnte Tank-Benutzungsschnittstelle: Zu jedem Aspekt wird ein Ausschnitt aus dem Smalltalk-Quelltext gezeigt, der beispielhaft die textuelle Programmierung dieses Aspekts zeigt. Allerdings ist die Beschreibung anhand des Quelltextes nur für Leser/innen verständlich, die bereits (Smalltalk) programmiert haben. Einzelne Illustrationen machen deutlich, wieviel implementierungsübergreifender visuelle Darstellungen der Aspekte sind, und zwar, weil übergreifende Aspekte für das Modell gewählt wurden.

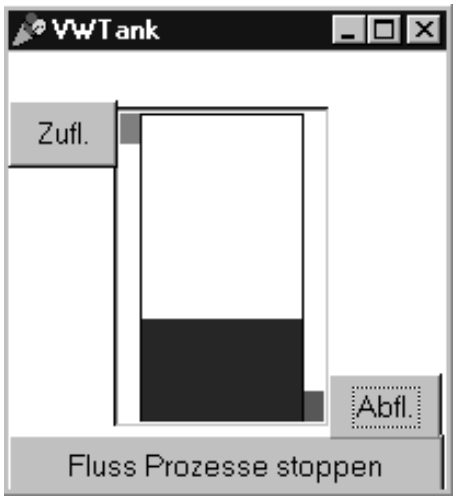
Jeder Aspekt kann durch verschiedene visuelle Methoden spezifiziert oder programmiert werden. Eine Auswahl visueller Methoden wird in Kapitel 6 vorgestellt.

5.2.1 Der Aspekt *Anordnung und graphische Attribute auf der Oberfläche* (*Layoutaspekt*)

Im Gegensatz zur Entwicklung anderer Softwaresysteme erlaubt die Benutzungsschnittstellen-Entwicklung durch die Anordnung auf der Oberfläche die Interaktion mit „wirklichen", d. h. wenn auch (bei klassischen zweidimensionalen graphischen Benutzungsschnittstellen) nicht anfaßbaren, so doch in ihrer tatsächlichen Form sichtbaren und interaktiv manipulierbaren Dingen.

Gerade bei technischen Anwendungen kann hier auch die physikalische Welt hinter dem Softwaresystem sichtbar gemacht werden.

Der Anordnungsaspekt ist intuitiv verständlich und für die Konstruktionsmethoden gut nutzbar. Im Beispiel ist die Anordnung folgendermaßen:



Die Elemente Tank (der wiederum aus den Elementen Begrenzung und Inhalt besteht) und Zu- bzw. Abfluß werden gemäß den physikalischen Gegebenheiten der tatsächlichen Anlage (z. B. Einfluß höher als Ausfluß) angeordnet.

Die Elemente, mit denen die Anwendung manipuliert werden kann (Knöpfe) werden in der Nähe der zu manipulierenden Elemente (Zu- und Abfluß) angeordnet. Die Motive (physikalische Gegebenheiten, optische Nähe) für die Anordnung können in anderen Aspekten (z. B. Ergonomie) der Benutzungsschnittstelle gefunden werden und sind nur im Rahmen der Unterstützung durch Muster (siehe Kapitel 4.4) Gegenstand dieser Betrachtungen.

Aus programmiertechnischer Sicht wird durch den Anordnungsaspekt nur festgelegt, auf welcher Position oder in welchem räumlichen Abhängigkeitsverhältnis zueinander sich die Elemente befinden. Im Tankbeispiel wird in der Methode `displayOn` beschrieben, wie die Größe und der Hintergrund gesetzt werden.

Der Tank selbst wird durch zwei Rechtecke beschrieben, die übereinander gezeichnet werden.

Die Position muß mithilfe der Größe des äußeren Polygons berechnet werden.

Programmbeispiel: Festlegung der Anordnung und der graphischen Attribute des Tanks

```
TankView methodsFor: 'displaying'

displayOn: aGraphicsContext

''Display the content of the Tank''

''Calculate border''
        | box |
        box := self bounds.

''Background''
        aGraphicsContext paint: ColorValue white.
        box displayFilledOn: aGraphicsContext.
''Display tank''
        aGraphicsContext paint: ColorValue blue.
        (box left @ (box bottom - (box height * model value / 100))
                    corner: box right @ box bottom)
        displayFilledOn: aGraphicsContext.
''Display border''
        aGraphicsContext paint: ColorValue black.
        box displayStrokedOn: aGraphicsContext.
''Display label''
        viewLabel
                ifTrue: [component name asComposedText
                         displayOn: aGraphicsContext]
```

Obwohl der Layoutaspekt eine Struktur von Elementen zeigt, ebenso wie der folgende Strukturaspekt, ist diese Art der Anordnung bei der Entwicklung von Benutzungschnittstellen von herausragender Bedeutung und wird deshalb nicht unter dem Strukturaspekt angeordnet.

5.2.2 Der Aspekt *Elemente und ihre Struktur - der Bauplan einer Benutzungsschnittstelle (Strukturaspekt)*

Dieser Aspekt beschreibt den Bauplan einer Benutzungsschnittstelle. Ein Vergleich mit der Disziplin Architektur: Bei der Betrachtung eines Hauses von außen ist die Fassade sichtbar, also die Oberflächenstruktur des Verputzes, die Farbe, die Anordnung der Fenster. Die Baupläne (z. B. Entwurfspläne und Ausführungspläne) hingegen zeigen und beschreiben die verschiedenen Materialien des Hauses und deren Aufbau, also die Bausteine und den Mörtel bzw. den Beton, die Isolierung, die Art des Verputzes, die innere Struktur des Hauses, also Zimmer, Mauern usw.

Statt einer Architektenzeichung des geplanten Aussehens des Hauses (entsprechend bei der Benutzungsschnittstelle der Layoutaspekt) benutzen die Bauleute immer die Baupläne bei der Ausführung, weil sie die Struktur des Hauses zeigen.

Genauso gehen auch die Ingenieurswissenschaften vor, wie bereits in Abschnitt 3.3.5 gezeigt: Der Aufbau eines Systems wird mit Hilfe verschiedener Ansichten beschrieben. Um in der Analogie zu bleiben: Was ist der Bauplan einer Benutzungsschnittstelle?

Prinzipiell bestehen objektorientierte Benutzungsschnittstellen, hinter der Fassade, aus Objekten und den Beziehungen zwischen ihnen. Die Objekte und die Beziehungen zwischen ihnen werden durch Klassen und ihre Struktur (Erbungshierarchien, Aggregationen, Assoziationen) beschrieben. Damit kann ein Klassendiagramm der Bauplan einer Benutzungsschnittstelle sein. Die Wahl, Objekte als Bausteine zu verwenden, ist allerdings willkürlich, genausogut könnten andere Datenstrukturen oder Rechnerarchitekturen verwendet werden. Außerdem können Informatiker/innen beliebige Objekte entwerfen: Architek/innen haben zwar die Wahl zwischen verschiedenen Materialien, z. B. Ziegel- oder Ytongsteine für das Mauerwerk, aber Informatiker/innen sind nicht an physikalische Gegebenheiten gebunden und können, um im Vergleich zu bleiben, ganze Wände spezifizieren oder auch ein Objekt, das ein Fenster inklusive Rahmen beschreibt[1]. Beim Tankbeispiel kann der Tank als eine Einheit, d. h ein Objekt betrachtet werden, das Zu- und Ablauf mitbeschreibt, er kann aber auch durch mehrere Einheiten, z. B. Behälter, Zu- und Ablauf, modelliert werden.

Darüberhinaus kann jede Einheit auch mehrere Repräsentationen haben, weswegen verschiedene Entwickler/innen verschiedene Baupläne entwerfen können.

Die Entwickler/in muß die möglichen Objekte und Strukturen kennen, um den konkreten Aufbau einer Benutzungsschnittstelle beschreiben zu können.

Über die Objektstruktur hinaus gehören zu dem Bauplan die Strukturen aller weiteren Konzepte, die Elemente beschreiben, aus denen sich die Benutzungsschnittstelle zusammensetzen kann: z. B. Software-Komponenten und ihre Struktur, wenn die Programmierumgebung Komponententechnologie anbietet, oder Widgets.

Beispiel: Ein möglicher Bauplan für die Elemente, aus denen der Tank aufgebaut ist

Das Programm für die Tankbenutzungsschnittstelle besteht aus den im folgenden dargestellten Elementen und ihren Strukturen. Die Klassifikation selbst spiegelt die grundlegende Modularisierung von Seeheim-Modell und MVC-Ansatz wider. Für die graphische Notation wurden UML-ähnliche Diagramme gewählt. Wer die UML oder eine ähnliche Beschreibungssprache kennt, bemerkt, daß Klassendiagramme zur Erklärung hilfreich sind[2]. Da sie nicht integriert sind, fehlen allerdings noch wichtige Informationen, deren visuelle Darstellungen auf das nächste Kapitel verschoben sind.

Elemente und ihre Strukturen sind:

- Sichtbare Elemente, im Tankbeispiel das Fenster, in dem die Benutzungsschnittstelle dargestellt wird, das Rechteck für den Rahmen des Tanks, das Rechteck für den Inhalt,

[1] Eine der Herausforderungen beim objektorientierten Programmieren ist daher auch das Finden bzw. Festlegen von Objekten [Sie92].

[2] Das Kapitel ist wesentlich schwerer zu lesen, wenn die Diagramme nicht gezeigt werden. Interessierte Leser/innen können die Graphiken zum Test beim Lesen verdecken.

der Zu- und Abfluß, der Zu- und Abflußsteuerknopf, der Knopf, mit dem die Prozesse angehalten werden können. Die Anordnung der Elementen auf der Oberfläche ist schon im Zusammenhang mit dem Layoutaspekt behandelt worden. Es ist aber auch in einer baumartigen Struktur festgelegt, welches die umfassenden Objekte (engl. *container*) sind, und welche Objekte in ihnen angeordnet sind.

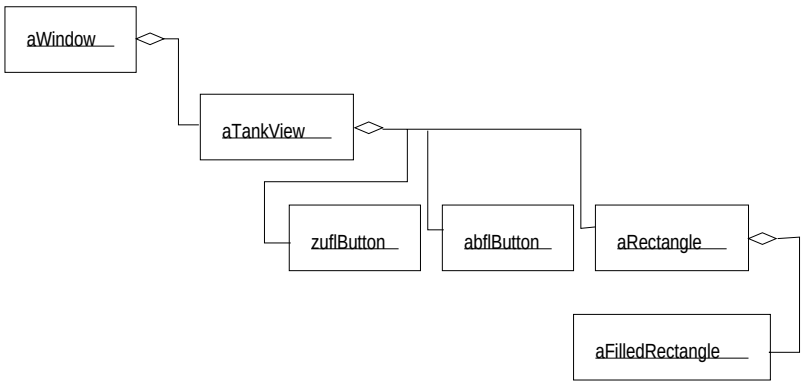


Abbildung 5.1: Aggregationsstruktur des Tankbeispiels

- Die Objekte werden durch Klassen beschrieben, die über eine Erbungsstruktur miteinander ebenfalls in einer Baumhierarchie verknüpft sind.

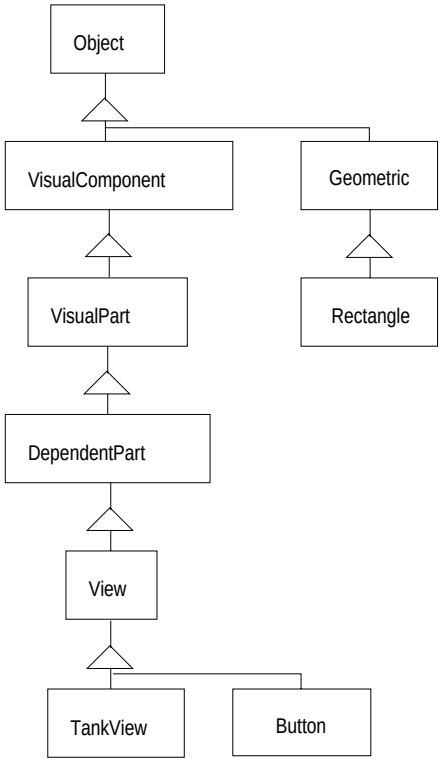


Abbildung 5.2: Erbungsstruktur der Klassen des Tankbeispiels

- Zusätzlich gibt es Objekte, die die sichtbaren Elemente verwalten oder das Verhalten vereinfachen. Z. B. können zwischen Objekten sog. Adapterobjekte geschaltet

werden, die Zugriffe auf Werte eines Objekts standardisieren, oder Objekte, die die Benutzungsschnittstellen-Objekte verwalten. Solche Objekte sind beim Betrachten einer Benutzungsschnittstelle nicht sofort sichtbar und müssen daher der Entwickler/in explizit bekannt sein. Auch sind sie von der Programmierumgebung abhängig, d. h. die Verwaltung wird in jeder Programmierumgebung anders organisiert. Der größte Teil der Zeit zum Erlernen einer Umgebung wird oft für das Verständnis dieser Objekte gebraucht.

- Nicht sichtbare Objekte, ebenfalls mit Aggregations- und Erbungshierarchie z. B.
  - Interaktionsobjekte (im Beispiel das Objekt `aStandardSystemController`).
  - Die eigentlichen Datenobjekte, die den Zustand der Anwendung widerspiegeln, oft Modellobjekte genannt (im Beispiel das Objekt `tank`).
  - Die Objekte, die zur Anwendung gehören (z. B. eine Sensordatenerfassung).
  - Auch die Objekte, die durch Klassen beschrieben werden, die durch Erbung mitbenutzt werden, müssen bekannt sein (z. B. die Klasse `View`).
- Weiterhin bieten Programmierumgebungen Abstraktions- und Kapselungsmechanismen an, die z. B. eine Verteilung des Softwaresystems erlauben. Ein solcher Mechanismus ist die Komponententechnologie. Obwohl die Benutzungsschnittstelle weiterhin aus Objekten besteht, setzt sie sich außerdem auch aus Komponenten zusammen. Komponenten werden in der Regel durch „Verdrahten", d. h. das Verbinden von Schnittstellen, strukturiert. Im Kapitel 4.4.3 wurde ein Komponentenmodell vorgestellt, das eine solche Kapselung einführt und weitere Elemente beschreibt, aus denen die Benutzungsschnittstelle dann auch zusammengesetzt werden kann.

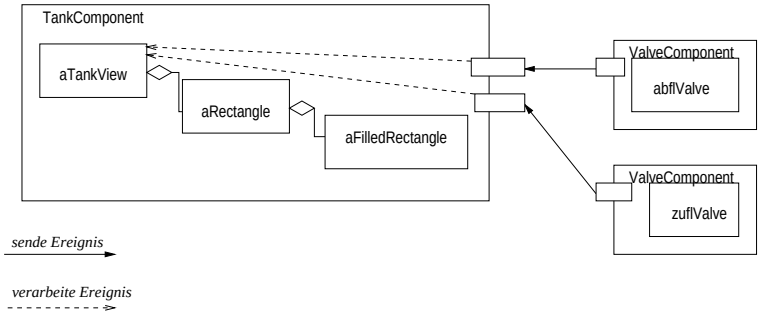


Abbildung 5.3: Der Beispieltank, aus Komponenten zusammengesetzt

Diese Vielfalt von Elementen und Strukturen, die je nach Programmierung verwendet werden können oder auch nicht, macht die Entwicklung von Benutzungsschnittstellen für Anfänger/innen schwierig.

So besteht bei erstem Hinsehen die Benutzungsschnittstelle für den Tank aus den sichtbaren Elementen.

Beim Ansehen des Programms[3] finden sich zunächst die Beschreibungen der Objekte, und zwar die Klasse `Tank` (als Datenobjekt) und `TankView` (als Objekt, das die Beschreibung des

[3]das hier aus Platzgründen und zur besseren Lesbarkeit nicht wiedergegeben wird,

sichtbaren Tankelements enthält). Alle sichtbaren Elemente selbst werden durch Ausprägungen von typischerweise ererbten Klassen wie `View` und `Rectangle` erzeugt.

Eine weitere Analyse des vollständigen Programms ergab, daß (im Beispiel, das mit VisualWorks implementiert ist) auch eine Klasse `VWTank` zur Verwaltung der sichtbaren Objekte existieren muß. Darüberhinaus muß z. B. die Klasse `ApplicationModel` als Oberklasse von `VWTank` bekannt sein. Eine Ausprägung der Klasse `StandardSystemController` wird als Interaktionsobjekt verwendet, usw. Solche Interna könnten durch eine angemessene visuelle Repräsentation vor der Entwickler/in verborgen werden, bzw. könnte eine visuelle Darstellung auch zum Verständnis solcher Strukturen führen.

5.2.3 Der Aspekt *Verhalten von Benutzungsschnittstellen* (*Verhaltensaspekt*)

„Das Verhalten" einer objektorientierten Benutzungsschnittstelle wird durch die Kommunikation der Oberflächenobjekte untereinander mit der Anwendung und mit der Benutzer/in festgelegt. Wie in Abschnitt 4.3.2 beschrieben, setzt es sich aus den verschiedenartigen Mechanismen der einzelnen Elemente zusammen. Dazu gehören bei objektorientierten Benutzungsschnittstellen:

- *Graphisches Verhalten:* Das Verhalten der Elemente auf der Oberfläche, und zwar in Abhängigkeit von den Interaktionen der Endbenutzer/in, von den internen Verarbeitungen der Oberflächenobjekte und von Berechnungen der Anwendung (der letzte Punkt wird jedoch als eigener Aspekt behandelt). Z. B. wird der Tank im Beispiel dynamisch gefüllt, er zeigt also „Füllverhalten". Ein anderes Beispiel sind die Knöpfe (engl. *buttons*), die von der Benutzer/in gedrückt werden können. Dabei wird, z. B. bei Zufluß und Abfluß, graphisch das Aussehen der Knöpfe während des Drückens geändert und das Füllverhalten des Tanks ändert sich (wenn der Abfluß zu ist, steigt der Füllstand ständig). Graphisches Verhalten kann auf allen Programmierebenen spezifiziert werden, im folgenden Beispiel zeigen die folgenden Programmbeispiele und sowie Abbildung 5.4 je eine Darstellung durch Basismechanismen (Ebene 0), bzw. Abbildung 5.4 auch eine aufgabenorientierte Beschreibung (Ebene 3).

Programmbeispiel: Textuelle Darstellung des Farbwechsels vom Abfluß, Ebene 0

```
[...] abfluss
‘‘wenn der Abfluss geoeffnet ist, hat der Knopf die Farbe gruen’’
ifTrue: [aGraphicsContext paint: ColorValue green]
‘‘wenn der Abfluss geschlossen ist, hat der Knopf die Farbe rot’’
ifFalse: [aGraphicsContext paint: ColorValue red].
```

Auch das Ändern der Farbe in Abhängigkeit vom Sauerstoffgehalt wird ähnlich

beschrieben:

Programmbeispiel: Textuelle Darstellung des Farbwechsels vom Abfluß, Ebene 0

```
displayOn: aGraphicsContext
...
‘‘Display tank’’
tank o2concentration
ifHigh:[aGraphicsContext paint: ColorValue blue.
(box left @ (box bottom - (box height * model value / 100))
corner: box right @ box bottom) displayFilledOn: aGraphicsContext.]
ifLow:[aGraphicsContext paint: ColorValue red.
(box left @ (box bottom - (box height * model value / 100))
corner: box right @ box bottom) displayFilledOn: aGraphicsContext.]
ifNormal:[aGraphicsContext paint: ColorValue green.
(box left @ (box bottom - (box height * model value / 100))
corner: box right @ box bottom) displayFilledOn: aGraphicsContext.]
```

Darstellung auf Ebene 3:

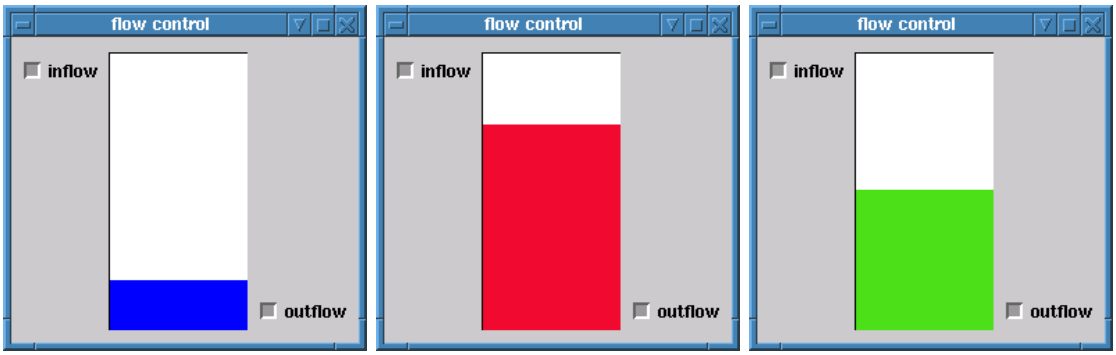


Abbildung 5.4: Zu hohe, zu niedrige und angemessene Sauerstoffkonzentration, aufgabenorientierte Darstellung (Ebene 3)

- *Objektorientiertes Verhalten*: Das Verhalten der Elemente wird bei Verwendung einer objektorientierten Programmierumgebung durch Methodenaufrufe programmiert. Diese sind (in der Regel allerdings auf eingebaute Grundoperationen aufbauend) die Basisblöcke objektorientierten Verhaltens. Grundlegendes Verhalten ist auch die Erzeugung von Objekten. Solche Basismechanismen gehören zu Ebene 0, die Spezifikation kann aber auch in anderen Ebenen erfolgen, die dann wieder auf Ebene 0 abgebildet werden müssen.

- *Verhalten durch Mechanismen der Klassenbibliothek* (Ebene 1): Dieses Verhalten basiert wieder auf dem objektorientierten Verhalten, jedoch auf einer höheren Abstraktionsstufe

als das objektorientierte Verhalten. Beispielsweise beruht der Abhängigkeitsmechanismus des MVC-Ansatzes auf Methodenaufrufen, wird aber als eigenständiges Verhalten gesehen.

5.2.4 Der Aspekt *Verbindung von Oberfläche und Anwendung* (*Anbindungaspekt*)

Dieser Aspekt wird wiederum gesondert betrachtet, da er die Grundlage von Modularisierungsansätzen für interaktive Systeme ist, wie dem Seeheim-Modell oder dem MVC-Ansatz. Wiederum beruht die Verbindung von Oberfläche und Anwendung auf dem Vorhandensein von nicht sichtbaren Objekten und Mechanismen, die diese Verbindung ermöglichen. Letztlich beschreibt dieser Aspekt, welche Teile der Anwendung auf der Oberfläche visualisiert werden und in welcher Form (und damit zum Teil auch das Verhalten auf der Oberfläche), und wie Signale von der Oberfläche (z. B. durch Interaktion der Benutzer/in) zur Anwendung gelangen. Dieser Aspekt wird z. B. in Smalltalk-Entwicklungsumgebungen explizit durch Mechanismen, wie dem Abhängigkeitsmechanismus, programmiert. Nur wer mit dem Abhängigkeitsmechanismus vertraut ist, versteht die Funktionsweise der Benutzungsschnittstelle, da die Verbindung im Programmcode nicht als eigener Aspekt in Erscheinung tritt, wie im folgenden Beispiel[4]:

Programmbeispiel: Textuelle Darstellung des Abhängigkeitsmechanismus (Ebene 1)

```
[...]
"Festlegung des Abhängigkeitsverhältnisses"
TankView model: tank.
[...]
"Benutzen des Abhängigkeitsmechanismus in der Klasse TankView"
update: [...].
‘‘Ein Aspekt des Modells hat sich geaendert. Normalerweise wird der Empfaenger
der Nachricht neu gezeichnet.
Subklassen koennen dieses Verhalten verfeinern.’’
self invalidate. ‘‘Benachrichtung zum Auffrischen des Bildschirms’’
self repairDamage.
```

5.2.5 Weitere Aspekte

In dieser Arbeit wird die Spezifikation der bereits beschriebenen Aspekte genauer betrachtet. Weitere Aspekte einer Benutzungsschnittstelle, für die - wie bereits erwähnt - jedoch keine Spezifikationsmethoden untersucht wurden, sind z. B.:

[4]Ich will mit diesem Codebeispiel nicht den Abhängigkeitsmechanismus erklären (ein gutes Lehrbuch für den Mechanismus in Smalltalk ist z. B. [Hop97]), sondern nur auf das Vorhandensein dieses Aspekts hinweisen.

- **Echtzeitaspekt**
  Dazu gehört z. B. Ein- und Ausgabe in Echtzeit.

- **Aspekt der Standardisierung und Richtlinien**
  Oft muß die Benutzungsschnittstelle gewissen Standards (wie Firmen- oder Werkzeugart-Standards) in Aussehen und Verhalten folgen. Dieser Aspekt kann durch Constraints auf dem Layout- bzw. Verhaltensaspekt realisiert werden.

- **Aspekt der Benutzungsfreundlichkeit**
  Hierfür gibt es ebenfalls Richtlinien. Das Anbieten von Hilfe-Werkzeugen fällt auch in diesen Aspekt.

5.2.6 Diskussion des Aspektmodells

In den vorhergehenden Abschnitten wurde gezeigt, in welche Aspekte eine Benutzungsschnittstelle „zerlegt" werden kann. Die Beschreibung der gesamten Benutzungsschnittstelle kann daher durch Beschreibung der einzelnen Aspekte erfolgen. An dieser Stelle soll noch einmal betont werden, daß es sich beim Aspektmodell um ein dem konzeptuellen Modell der Entwickler/in angepaßtes Modell handelt, daß der Wert also darin besteht, den Entwickler/innen eine klarere Vorstellung zu vermitteln. Alle (aus den in dieser Arbeit untersuchten) bekannten Programmierprinzipien lassen sich in einen der Aspekte einordnen, daher kann eine Benutzungsschnittstelle durch Spezifikation der Aspekte vollständig programmiert werden, wie auch in den darauffolgenden Kapiteln in der Implementierung deutlich wird.

Die verschiedenen (Teil-)Aspekte sind jedoch nicht orthogonal, z. B. wird eine Spezifikation des Objektverhaltens auch einen Teil des Oberflächenverhaltens oder die Anbindung der Oberfläche an die Anwendung spezifizieren. Wie die Beispiele in Abschnitt 7 zeigen, kann durch Spezifikation aller Aspekte (im Rahmen der Anforderungen) eine vollständige Benutzungsschnittstelle spezifiziert werden, aber die Spezifikationen sind teilweise redundant (durch die Reihenfolge der Generierung von Programmtext jedoch widerspruchsfrei).

Bewertung des Modells

Im folgenden dienen die Aspekte *Layout*, *Struktur*, *Verhalten* und *Anbindung* als Grundlage für die Entwicklung eines Werkzeugs. Mit Methoden zur Spezifikation dieser Aspekte können Benutzungsschnittstellen, die den Anforderungen aus Kapitel 2 und 3 genügen (zweidimensionale Benutzungsschnittstellen für technische Systeme), realisiert werden.

Das Aspektmodell ist implementierungsunabhängig und kann daher als Metamodell zur Beschreibung von sprachabhängigen Mechanismen verwendet werden.

Es beschreibt Benutzungsschnittstellen auf allen Entwicklungsebenen (textuelle Programierung (Ebene 0) bis aufgabenorientierte visuelle Programmierung (Ebene 3)), und eine Benutzungsschnittstelle kann daher mit Hilfe von Spezifikationsmethoden auf allen vier Ebenen konstruiert werden. Dieser Vorteil wird in Abschnitt 5.3 weiter ausgeführt.

Mit dem Aspektmodell wird auch das einleitend erwähnte Ziel erreicht, (visuelle) Spezifikationsmethoden möglichst einfach zu halten: Indem jede (visuelle) Spezifikationsmethode nur einen Aspekt definieren muß, sinkt z. B. die Anzahl der benötigten Symbole (die sog. Diffusion [GP96]).

Durch das Aspektmodell können (visuelle) Spezifikationsmethoden miteinander integriert werden. Wie in Kapitel 3 beschrieben, arbeiten Ingenieur/innen bei der Konstruktion mit verschiedenen Spezifikationsmethoden. Diese bekannte Arbeitsweise wird durch das Aspektmodell somit ebenfalls unterstützt, wenn es gewünscht wird. Personen, die lieber mit ausschließlich einer Darstellung arbeiten, können weiterhin auf die textuelle Programmierung zurückgreifen, wenn die Implementierung modellbasiert erfolgt, wie in Kapitel 7 beschrieben.

Abgrenzung zu anderen Ansätzen

Karsai [Kar95] schlägt vor, modellbasierte Prozeßautomatisierungssysteme zu entwickeln, bei denen ebenfalls jedes einzelne Modell einen Aspekt repräsentiert, um die Komplexität der Darstellung zu reduzieren. Es wird jedoch nicht die Benutzungsschnittstelle konstruiert, sondern die Anwendungsfunktionalität. Bei den von ihm betrachteten Aspekten handelt es sich z. B. um Echtzeitaspekte, mathematische Aspekte von Prozeßmodellen etc. Mehrere visuelle Darstellungen können wiederum ein Modell beschreiben, und mehrere Modelle können einen Aspekt definieren.

Objektorientierte Analyse- und Designmodelle

Die Aspekte *Struktur* und *Verhalten* objektorientierter Programme sind auch Gegenstand von Modellierungsmethoden wie den objektorientierten Analyse- und Designmethoden. Durch das Anwendungsfeld *Entwicklung von Benutzungsschnittstellen* können jedoch in das vorgestellte Aspektmodell benutzungsschnittstellen-spezifische Aspekte, z. B. *Layout* und die *Anbindung*, eingebracht werden. Auf einer abstrakteren Ebene handelt es sich jedoch um den gleichen Ansatz zur Modularisierung von objektorientierten Softwaresystemen. Dies wird auch im nächsten Kapitel bei den Methoden zur Visualisierung der Aspekte deutlich, da beide Ansätze z. B. für den *Struktur-Aspekt* dieselben visuellen Beschreibungsmethoden verwenden.

Aspektorientiertes Programmieren

Seit einiger Zeit gibt es den Ansatz des sog. aspektorientierten Programmierens [KLM+97]. Bei der aspektorientierten Programmierung werden ebenfalls Aspekte definiert, die dann in die Programmstruktur (Methoden, Module etc.) automatisch eingewoben werden (durch sog. *Aspektweber*, engl. *aspect weaver*). Für jeden Aspekt gibt es dabei ein Programmiersprachenkonstrukt, mit dem genau dieser Aspekt ausgedrückt werden kann. Die zugrundeliegende Idee ist also sehr ähnlich, aber sie wird verschieden umgesetzt: Während bei meinem Aspektmodell der Aspekt zunächst durch eine Visualisierung ausgedrückt wird, die dann in *allgemeine* Konstrukte der Implementierungssprache umgesetzt wird, werden beim aspektorienten Programmieren *neue* Sprachkonstrukte eingeführt, die nur zur Definition dieses Aspekts dient. Das hier verwendete Aspektmodell könnte durch aspektorientiertes Programmieren unterstützt werden, indem Sprachkonstrukte und ein Aspektweber für die vorgestellten Aspekte entworfen würde. Dann würde die Visualisierung auf aspektspezifische Sprachkonstrukte abgebildet werden.

5.3 Ebenenübergreifende Modellierung mit dem Aspektmodell: Falltüren zwischen den Programmierebenen 3 bis 0

In diesem Abschnitt wird die Diskussion über Lern- und Verständnisbarrieren von Seite 24 wieder aufgegriffen. Mit dem Aspektmodell können die von Myers [Mye00a] kritisierten und bereits diskutierten Lernhürden abgemildert werden; es erlaubt ein die Programmierebenen übergreifendes Verständnis von Benutzungsschnittstellen.

Auf jeder der definierten Programmierebenen

- textuelle Programmierung der durch die Sprache vorgegebenen Mechanismen (Ebene 0),
- textuelle Programmierung der durch die Klassenbibliothek gegebenen Strukturen und Mechanismen (Ebene 1),
- visuelle Programmierung der durch die Klassenbibliothek gegebenen Strukturen und Mechanismen (Ebene 2) und
- aufgabenorientierte Programmierung (Ebene 3)

wird deutlich, welche Aspekte mit einer vorliegenden visuellen oder textuellen Methode definiert werden können.

Wenn eine solche Methode auf einer Ebene nicht hilft, die Vorstellungen der Entwickler/in umzusetzen, kann eine „Falltür" [SBG00] zur nächst tieferen Ebene geöffnet werden. Es wird dann eine Methode angeboten, die auf der tieferen Ebene genau diesen Aspekt beschreibt. Damit muß nur ein Teil der Lernhürde auf einmal überwunden werden, und die Lernkurve wird damit glatter und angenehmer zu „ersteigen".

Als Beispiel sei die Programmierung des MVC-Prinzips genannt. Ein Teil der Programmierung des Verhaltensaspekts einer Benutzungsschnittstelle kann durch aufgabenbezogene Visualisierung (Ebene 3) erfolgen, z. B. mit der später in Abschnitt 6.3.1 beschriebenen Verhaltensbibliothek.

Vielleicht ist jedoch ein gewünschtes Verhalten in dieser Bibliothek nicht vorhanden. Dann muß die Programmierer/in auf die Ebene 2 „heruntersteigen" und das Verhalten mittels des MVC-Musters programmieren. Dazu dient z. B. der später in Abschnitt 6.3.4 beschriebene MVC-Editor, der visuelles Programmieren einer MVC-Einheit erlaubt. In Abschnitt 7.4 wird ein solcher vertikaler Abstieg anhand einer Fallstudie bei der Programmierung des Tankbeispiels genauer erläutert.

Das Konzept *Definition des Verhaltensaspekts* wird von Ebene 3 in Ebene 2 abgebildet, d. h. die Entwickler/in kann jederzeit anhand des Aspektmodells die Beziehung zwischen den beiden Spezifikationsmethoden erkennen und das Konzept des MVC-Musters leichter erlernen. Beim weiteren Absteigen, bis hin zur rein textuellen Programmierung, können solche konzeptuellen Modelle „mitgenommen" werden.

Für Anfänger/innen in der (Benutzungsschnittstellen-)Programmierung ist daher das folgende Vorgehen möglich:

1. Verstehen des Aspektmodells.
2. Visuelle Spezifikation mit aufgabenbezogenen Methoden (Ebene 3).
3. Wenn diese nicht ausreichen:
   (a) Visuelle Spezifikation mit Komponenten (Abstieg zu Ebene 2),
   (b) visuelle Spezifikation mit Hilfe von Entwurfsmustern (ebenfalls Ebene 2) oder
   (c) visuelle Spezifikation sonstiger Mechanismen (Ebene 2).
4. Wenn diese nicht ausreichen: Textuelle Spezifikation mit Entwurfsmustern (Abstieg auf Ebene 1)
5. Normale textuelle Programmierung (Abstieg auf Ebene 0).

Abbildung 5.5 skizziert die Zusammenhänge zwischen den Programmierebenen und dem Aspektmodell am Beispiel der Verhaltensmodellierung des Tankbeispiels:

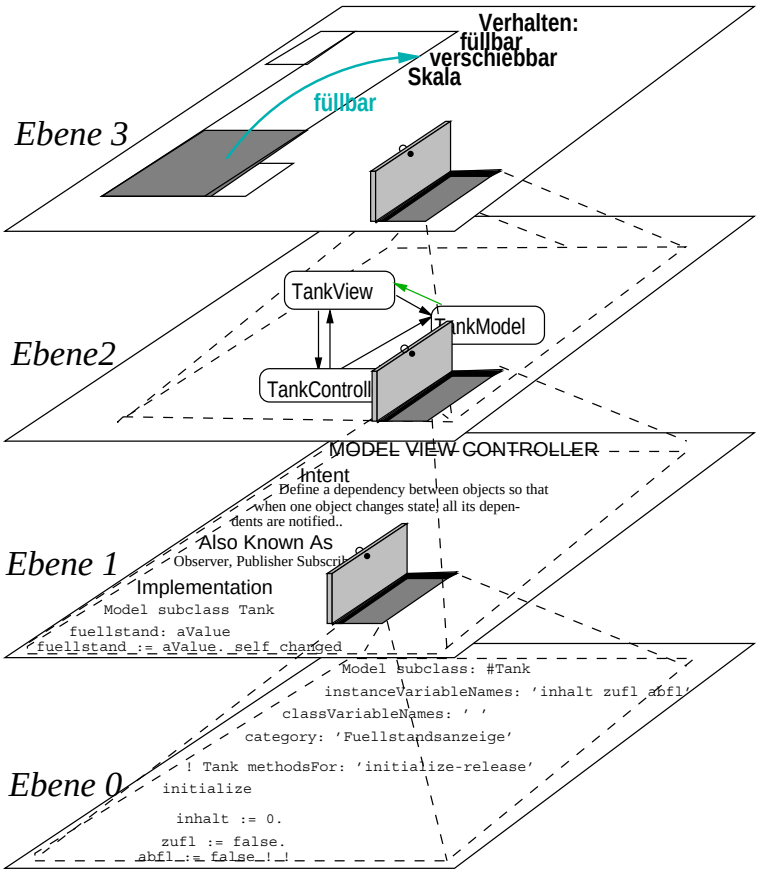


Abbildung 5.5: Modellierung des Verhaltensaspekts auf verschiedenen Ebenen

Kapitel 6

Visuelle Methoden zur Spezifikation der Aspekte

At the end of the day, there has to be a program.
So I nominate coding as the one activity
we know we can't do without.
Whether you draw diagrams that generate code
or you type it at a browser, you are coding. [Bec00]

In diesem Kapitel werden Beispiele für visuelle Spezifikationsmethoden vorgestellt, mit denen die im letzten Kapitel beschriebenen Aspekte *Layout, Struktur, Verhalten* und *Anbindung* einer Benutzungsschnittstelle spezifiziert werden können. Einige Spezifikationsmethoden wurden prototypisch durch das Werkzeug COMBO implementiert. Die Benutzungsschnittstelle und Implementierungsdetails von COMBO werden im nächsten Kapitel beschrieben.

Einführung

Visuelle Spezifikationsmethoden

Viele visuelle Notationen können, mit einer passenden Interpretation der Graphik, als Spezifikationsmethoden verwendet werden. Bestimmte Klassen von visuellen Notationen hat Myers in [Mye90] als geeignet für visuelle Spezifikationsmethoden klassifiziert:

- Flußdiagramme (Programmablaufpläne)
- Flußdiagramm-Abkömmlinge (z. B. Petrinetze)
- Datenflußdiagramme
- Gerichtete Graphen
- Graphen-Abkömmlinge
- Blockdiagramme
- Matrizen
- Formulare
- Sätze, gebildet aus Piktogrammen
- Tabellenkalkulation
- Spezifikation durch Vormachen

Auswahl und Beschreibung der in dieser Arbeit verwendeten visuellen Spezifikationsmethoden

Die Auswahl der in diesem Kapitel vorgestellten visuellen Spezifikationsmethoden wurde nach folgenden Kriterien getroffen:

- Für jeden der Aspekte *Layout*, *Struktur*, *Verhalten* und *Anbindung* sollte mindestens eine visuelle Spezifikationsmethode vorgestellt werden. Damit soll gezeigt werden, daß das Konzept umsetzbar ist, eine Benutzungsschnittstelle im Aspektmodell mit Hilfe solcher visuellen Methoden zu definieren (Aspektansatz).

- Der Schwerpunkt bei der Entwicklung (und Implementierung) der visuellen Spezifikationsmethoden liegt auf der Visualisierung von Entwurfsmustern, HCI-Mustern und Komponenten, die in Abschnitt 4.5 vorgestellt wurden.

- Der in Kapitel 5 beschriebene „Falltürenansatz" soll durch Methoden auf jeder Ebene in mindestens einem Aspekt verdeutlicht werden. Hierfür wurde der Verhaltensaspekt mit Implementierung durch den MVC-Ansatz gewählt.

Jede der im Rahmen dieser Arbeit untersuchten Methoden kann sehr ausführlich diskutiert werden, was den Umfang der Arbeit jedoch sprengen würde. Um die Beschreibung der visuellen Spezifikationsmethoden möglichst kurz zu halten, wird folgendes Schema benutzt:

Schema zur Beschreibung jeder einzelnen visuellen Spezifikationsmethode

Name der Spezifikationsmethode

⋆ ⋆ ⋆

A. Einführung und Notation
Eine kurze Beschreibung, evtl. mit Verweis auf die Literatur. Die Notation, Vorgehensweise oder Interaktionsform wird vorgestellt, falls sie nicht aus der Literatur bekannt ist; ebenso erfolgt eine Beschreibung der visuellen Prinzipien gemäß Definition in Kapitel 2.

⊙ ⊙ ⊙

B. Konzeptuelles Modell und Programmierebene
Welche Vorstellung liegt der Spezifikationsmethode zugrunde? Auf welcher Programmierebene erfolgt die Spezifikation?

⋈⋈⋈

C. Besonderheiten und Bewertung
Was macht diese Spezifikationsmethode besonders geeignet für die Definition eines Aspekts. Wie hat sich die Methode bewährt?

∼∼∼

D. Realisierung im Werkzeug COMBO
Überblick über die Implementierung und Anwendung in COMBO.

Reihenfolge der vorgestellten Methoden

Die Spezifikationsmethoden werden in der folgenden Reihenfolge vorgestellt:

- Zuerst werden nach den genannten Kriterien ausgewählte visuelle Spezifikationsmethoden für die in Kapitel 5 gezeigten *Aspekte* des Aspektmodells beschrieben. Die Auswahl soll dabei als Vorschlag verstanden werden; weitere Spezifikationsmethoden können das in dieser Arbeit vorgestellte Vorgehen ergänzen. (Ich hoffe, daß die Leserinnen und Leser durch diese Arbeit zum Entwickeln weiterer Methoden angeregt werden.)

- Daran anschließend werden die in dieser Arbeit entwickelten visuellen *Spezifikationsmethoden für Konzepte* wie *Anwendungsrahmen*, *Entwurfsmuster* und *Komponenten* erläutert. Diese Methoden werden deshalb in eigenen Abschnitten vorgestellt, weil sie mehrere Aspekte oder Programmierebenen zusammenfassen und das Aspektmodell nur benutzen. D. h., obwohl Anwendungsrahmen, Entwurfsmuster und Komponenten mit den Notationen der visuellen Spezifikationsmethoden der einzelnen Aspekte spezifiziert werden, spezifizieren sie Teile mehrerer Aspekte gleichzeitig.

  - Visuelle Spezifikationsmethoden für *Anwendungsrahmen* beschreiben mehrere Aspekte. Sie werden in dieser Arbeit jedoch aufgrund ihrer Ähnlichkeit zu den Software Engineering-Entwurfsmustern nur kurz diskutiert.

  - *Entwurfsmuster* beschreiben ebenfalls mehrere Aspekte. Insbesondere Entwurfsmuster aus dem Bereich Softwareentwicklung werden meist mit Hilfe der UML beschrieben, die für objektorientierte Software ebenfalls mehrere visuelle Beschreibungen für Klassenstrukturen und Objektverhalten anbietet.

  - Die *Komponententechnologie* dagegen erfordert neue Überlegungen, da jede Komponente alle Aspekte in sich vereint und darüberhinaus das Zusammenarbeiten und die Kommunikation der Komponenten neue Mechanismen einführt. Die Spezifikation einer einzelnen Komponente kann zunächst mithilfe der verschiedenen visuellen Methoden für die einzelnen Aspekte erstellt werden. Für die Kapselung und das Zusammensetzen von Komponenten wurden im Rahmen dieser Arbeit jedoch neue visuelle Spezifikationsmethoden entwickelt.

Für jeden Aspekt wird zunächst eine Übersicht *prinzipiell in Frage kommender Methoden* und eine kurze Abgrenzung zum Stand der Forschung gegeben, danach werden einige Methoden

genauer vorgestellt, die im Rahmen dieser Arbeit untersucht und integriert wurden:

Abschnitt 6.1 Visuelle Spezifikationsmethoden für den Aspekt ***Layout***
- Anordnungswerkzeug
- Zeichenprogramme
- Layoutbibliothek

Abschnitt 6.2 Visuelle Spezifikationsmethoden für den Aspekt ***Struktur***
- Klassen- und Objektdiagramme

Abschnitt 6.3 Visuelle Spezifikationsmethoden für den Aspekt ***Verhalten***
- Verhaltensbibliothek
- Objekt-Petrinetze
- visuelle Methodenbeschreibung
- visuelle Spezifikation der Anwendung des MVC-Musters

Abschnitt 6.4 Visuelle Spezifikationsmethoden für den Aspekt ***Anbindung***
- Objekt-Petrinetze
- visuelle Methodenbeschreibung
- Auswählen und Beschreiben
- Verhaltensbibliothek

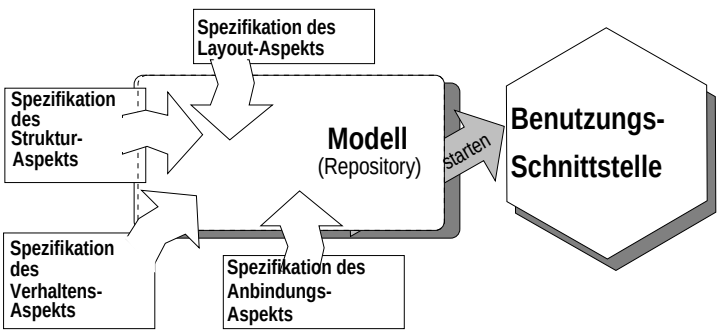
Abschnitt 6.5 Visuelle Spezifikationsmethoden für ***Anwendungsrahmen***
- Visualisierung der Entwurfsmuster in Anwendungsrahmen

Abschnitt 6.6 Visuelle Spezifikationsmethoden für ***Entwurfsmuster***
- Entwurfsmuster-Assistent
- Entwurfsmuster-Diagrammeditor
- Model-View-Controller-Editor

Abschnitt 6.7 Visuelle Spezifikationsmethoden für ***Komponenten***
- visuelles Entwerfen von Komponenten
- visuelles Zusammensetzen von Komponenten

Architektur zum Verwenden visueller Spezifikationsmethoden im Werkzeug COMBO

COMBO unterstützt die visuellen Spezifikationsmethoden durch eine Architektur, die die Ergebnisse der Methoden miteinander integriert:



Die Gesamtspezifikation der Benutzungsschnittstelle wird aus der Spezifikation der einzelnen Aspekte zusammengestellt, wie in der nebenstehenden Abbildung zu sehen ist. Aus dieser Spezifikation könnte der Quellcode generiert werden. Durch die Wahl von Smalltalk als Implementierungssprache (siehe Kapitel 7) entfällt aber dieser Schritt, und aus dem Modell kann die Laufzeit-Benutzungsschnittstelle unmittelbar gestartet werden.

Abbildung 6.1: Basisarchitektur von COMBO

Jede einzelne der Aspekt-Spezifikationen erfolgt durch Editoren, mit denen die visuelle Spezifikation erstellt werden kann. Abbildung 6.2 zeigt, welche Editoren zur Spezifikation der einzelnen Aspekte verwendet werden können.

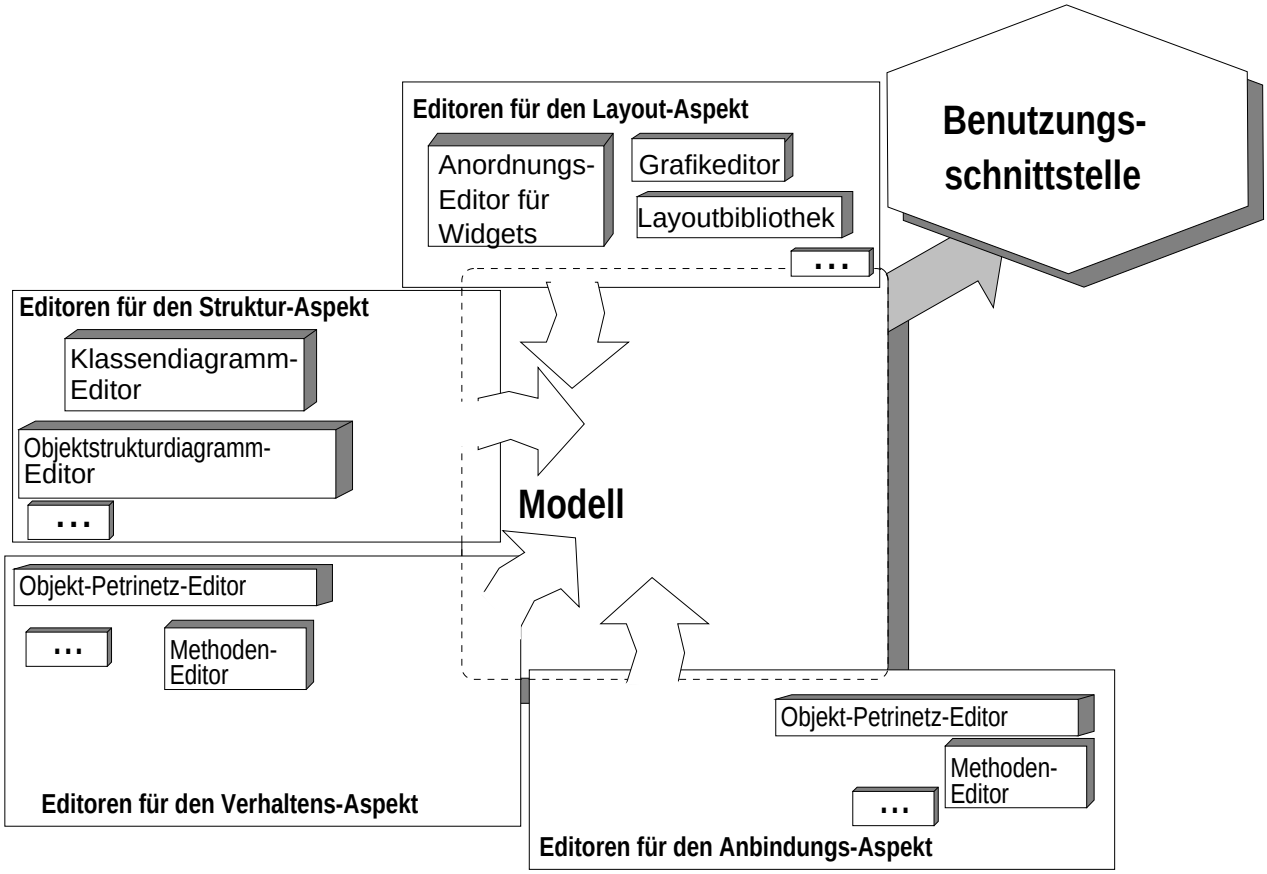


Abbildung 6.2: Editoren zur Spezifikation der Aspekte

Es gibt in COMBO auch visuelle Spezifikationsmethoden, die keine eigenen Editoren anbieten, sondern auf den anderen Editoren arbeiten, z. B. „Auswählen und Beschreiben“ für den Anbindungsaspekt.

Weiterhin gibt es visuelle Spezifikationsmethoden, die unterstützend für andere Methoden sind, z. B. die Editoren für Software Engineering-Entwurfsmuster oder die Komponenten-Editoren. Abbildung 6.3 zeigt alle Editoren.

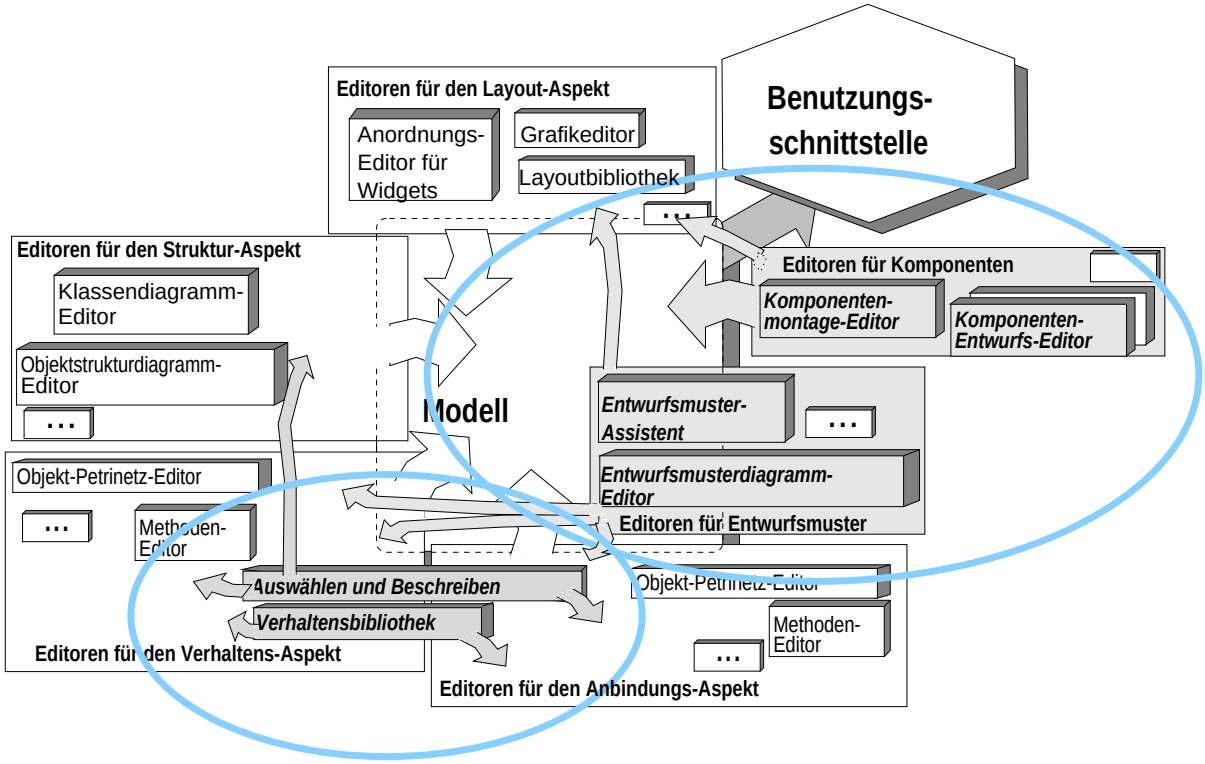


Abbildung 6.3: Aspektunterstützende Spezifikationsmethoden: Entwurfsmuster und Komponenten

6.1 Der *Layoutaspekt*: Anordnung auf der Oberfläche und graphische Attribute

Übersicht über Spezifikationsmethoden für das Layout

Das Layout einer Benutzungsoberfläche, also die Anordnung der graphischen Elemente der Benutzungsschnittstelle, ist durch die Sichtbarkeit seiner Bausteine sehr gut auf der aufgabenorientierten Ebene (Ebene 3) zu erstellen.

Ein Beispiel dafür ist die Klasse der **Anordnungswerkzeuge** (engl. ***interface builder***), seltener werden **Zeichenprogramme** zur Gestaltung der Oberfläche benutzt, da sie in bestehenden Werkzeugen schwierig mit den Widgets zu integrieren sind.

Neben solchen *Anordnungen von Hand* gibt es prinzipiell auch die *automatische Anordnung*, z. B. bei automatischer Generierung der Oberfläche aus Datenstrukturen (z. B. im Werkzeug

Janus, siehe [Bal93]). Solche Werkzeuge werden hier aber nicht weiter betrachtet, da sie die Anordnung nicht visuell beschreiben.

Zur Unterstützung werden bei Anordnungswerkzeugen **Formatierungshilfen** verwendet, die Gruppierungen und Ausrichtungen der Gruppierungen erlauben.

Dazu zählen auch halbautomatische Layoutstrategien wie die sog. *Layoutmanager* [Hen97], die in der Java-Klassenbibliothek verwendet werden, oder die *Gestalter* im HotDoc-Anwendungsrahmen [Buc98]. Eine solche Formatierungshilfe gibt durch eine auswählbare Strategie die relative Positionierung der Elemente auf der Oberfläche an. Die absoluten Positionen werden je nach gewählter Strategie berechnet. Beispiele für solche Layoutstrategien sind Formatierungen wie „Anordnung von links nach rechts in Reihen" oder „Anordnung jedes Widgets auf einer eigenen Karteikarte". Layoutstrategien unterscheiden sich von vollautomatischen Ansätzen dadurch, daß nur eine grobe Anordnung als Bedingung vorgegeben wird.

Denkbar sind auch Formatierungshilfen, die die Integration der Inhalte von Stilführern oder Richtlinien in ein bestehendes Layout durch eine visuelle Darstellung unterstützen.

Im Rahmen dieser Arbeit werden **anwendungsspezifische Anordnungen** durch die Layoutbibliothek berücksichtigt.

Graphische Constraints (dts. einschränkende Bedingung, zeichnerische Nebenbedingungen [Hof97]) können ebenfalls zur Anordnung von Elementen auf der Oberfläche wie auch zur Definition von Verhalten verwendet werden. Ein Constraint-System zur Anordnung graphischer Elemente erlaubt das deklarative Festlegen graphischer Eigenschaften und automatisches Sicherstellen der Konsistenz bei Änderung der Graphik (nach [Szw92, Ege92]).

Dahinter steckt die Vorstellung, Beziehungen zwischen Elementen durch Beschreibung der Zusammenhänge auszudrücken, anstatt bezogen auf ein Grundsystem. Beispielsweise wird die Position eines graphischen Benutzungsschnittstellen-Elements nicht durch Werte in einem Koordinatensystem ausgedrückt, sondern durch Beziehungen der Form: „Der Knopf ist auf der gleichen Höhe wie der Knopf links daneben". Solche Beziehungen lassen sich auch visuell ausdrücken: In den Werkzeugen *Peridot* [Mye88] und *Lapidary* [Mye93] können graphische Bedingungen visuell durch Dialogboxen und durch Vormachen deklariert werden (auch das Verhalten). *Rockit* [KLW92] leitet mögliche Bedingungen aus den Interaktionen der Benutzer/in ab. Die Programmierung erfolgt sowohl bei Rockit als auch bei Lapidary aufgabenorientiert (Ebene 3). Die visuelle Spezifikation von graphischen Constraints ist für ein Werkzeug wie COMBO sehr interessant, wurde aber nicht umgesetzt, weil es bereits viele Implementierungen gibt, und die Programmierung mit Constraints die Einführung eines weiteren Mechanismus bedeutet hätte.

Eine interessante Variante ist das interaktive Skizzieren, bei dem Skizzen, die mit einem Eingabestift gemacht werden, als Grundlage für die Anordnung dienen. Ein solches System ist z. B. SILK [LM00], dessen Techniken auch in DENIM [LNHL00] zum Entwurf von Webseiten verwendet werden.

6.1.1 Anordnungswerkzeuge (*interface builder*)[1]

⋆ ⋆ ⋆

A. Einführung und Notation

Anordnungswerkzeuge benutzen das Anordnungsprinzip wie in Abschnitt 2.3 beschrieben, i. d. R. zusammen mit der Möglichkeit einer Auswahl von Elementen, die in einem Feld angeordnet werden können. Die meisten Programmierumgebungen bieten heute ein solches Werkzeug an.

Erlaubt ist eine beliebige zweidimensionale Anordnung aller Elemente, die zur Auswahl stehen. Diese werden auf Objekte als Ausprägungen von Klassen der Klassenbibliothek abgebildet, deren Attribute (z. B. Position, Farbe, Markierung) so gesetzt sind, wie sie bei der Anordnung von der Entwickler/in definiert wurden.

⊙ ⊙ ⊙

B. Konzeptuelles Modell und Programmierebene

Die Spezifikation des Layouts erfolgt durch Anordnen, d. h. aufgabenorientiert (Ebene 3). Die Vorstellung ist, daß Elemente (z. B. ein Knopf) aus einer Auswahl von bereits beschriebenen Klassen von Elementen ausgeprägt werden kann. Dabei werden seine Attribute explizit belegt (z. B. bei einem Knopf das Attribut `beschriftung` durch die Zeichenreihe `''Fluss Prozesse stoppen''`) . Abbildung 6.4 zeigt ein typisches Beispiel.

⋈ ⋈ ⋈

C. Besonderheiten und Bewertung

Anordnungswerkzeuge sind sehr intuitiv: Einerseits zeigen sie alle in der Umgebung vorhandenen Elemente, die sich die Entwickler/in nicht mehr merken muß, andererseits simulieren sie übliche Anordnungsstrategien (z. B. beim Layout einer Zeitung).

Anordnungswerkzeuge verbergen zwei Eigenschaften objektorientierter Programme, deren Sichtbarkeit die Komplexität des konzeptuellen Modells erhöhen würden:

1. Den Unterschied zwischen Klassen und Ausprägungen und
2. welcher Art die anzuordnenden Elemente sind,

d. h. ob es sich um Objekte, Komponenten oder Datenstrukturen der strukturierten Programmierung handelt, merkt die Entwickler/in nicht.

∼ ∼ ∼

D. Realisierung im Werkzeug COMBO

COMBO bietet zur Zeit zwei Anordnungswerkzeuge an: Eines für „normale" Widgets und eines für die Entwicklung des Aussehens von einzelnen Komponenten.

Anordnungswerkzeug für Widgets:

Statt einer Eigenentwicklung wurde das in der Entwicklungsumgebung VisualWorks enthaltene Anordnungswerkzeug UIBuilder erweitert.

[1] Für diese und die folgenden Methoden wird das in Kapitel 6 eingeführte Schema benutzt.

Die Auswahl der Widgets erfolgt mit Hilfe einer verkleinerten Darstellung aller vorgefertigten Widgets auf einer Palette (siehe Abbildung 6.4).

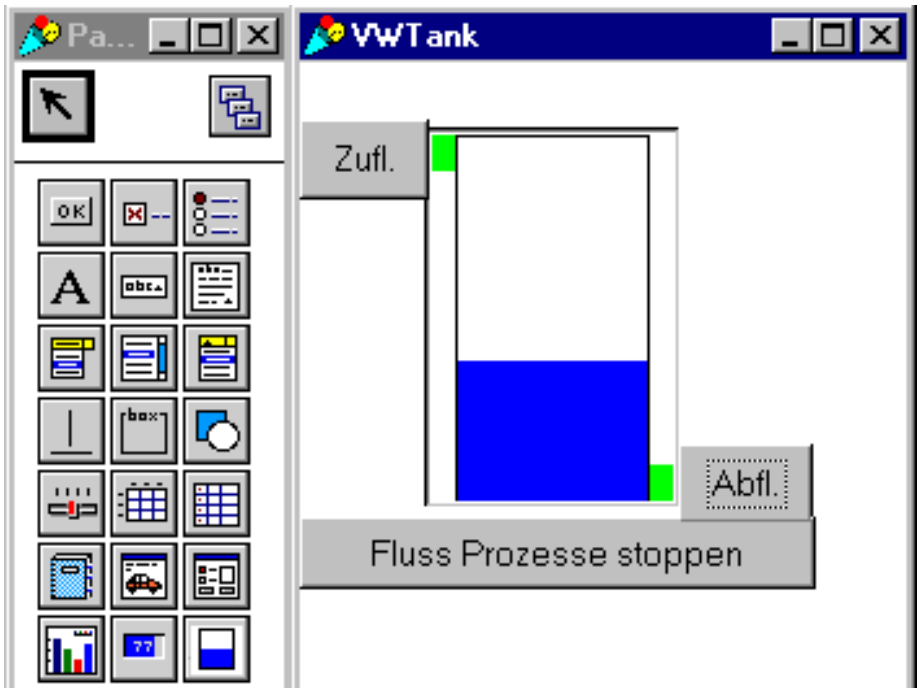


Abbildung 6.4: Anordnung aus Tankwidget und Knöpfen

Anordnungswerkzeug für das Aussehen von Komponenten:
Komponenten werden zur Zeit mit einem eigenen Anordnungswerkzeug, genannt LCLComponent [Mar00], entwickelt. Das ist in der prototypischen Implementierung nötig, um Komponenten kapseln zu können. Ist eine Komponente fertig entwickelt, kann sie der Palette als Widget hinzugefügt werden.

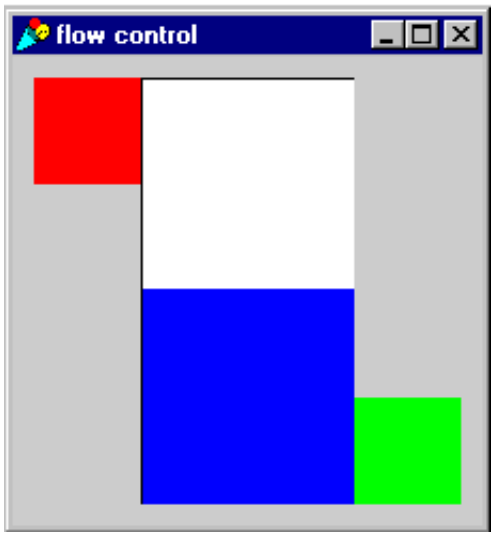


Abbildung 6.5: Anordnung innerhalb der Tank-Komponente

6.1.2 Zeichenprogramme

⋆ ⋆ ⋆

A. Einführung und Notation

Zeichenprogramme wie CorelDraw [Cor00], Designer [MIC98] oder auch die bekannten Zeichenfunktionen in Microsoft Word oder PowerPoint werden hauptsächlich zum Erstellen von Dokumenten verwendet.

Mit den graphischen Grundelementen eines Zeichenprogramms (z. B. Linien, Rechtecke, Kreise, Freihandzeichnungen) kann auch das Aussehen von Elementen der Benutzungsschnittstelle festgelegt werden. Dazu muß es prinzipiell möglich sein, einzelne graphische Elemente oder Gruppierungen von Elementen auf Widgetklassen abzubilden. Während bei den Anordnungwerkzeugen nur Ausprägungen der Klassen erzeugt werden, müssen die Zeichnungen zweifach interpretiert werden: Aus der Zeichnung, oder Teilen davon, müssen

1. Widgetklassen erzeugt werden, beispielsweise als Unterklassen einer Klasse, die in eine Benutzungsschnittstelle eingebunden werden. Es ist hilfreich, wenn diese Klassen dann
2. auch als Widgets in einem Anordnungswerkzeug oder einem anderen Layoutwerkzeug verwendet werden können. Dies ist z. B. im Werkzeug XFaceMaker [Con94] mit einem speziellen Einbindungsmechanismus möglich.

⊙ ⊙ ⊙

B. Konzeptuelles Modell und Programmierebene

Das Zeichnen eines Widgets entspricht der Vorstellung, daß sein Aussehen aus graphischen Basiselementen (Kreis, Rechteck) zusammengesetzt ist. Wenn keine passenden vordefinierten Elemente vorhanden sind, ist das Zeichnen die einfachste aufgabenorientierte Möglichkeit, das Aussehen festzulegen.

Die meisten Zeichenprogramme erlauben direkte Manipulation der graphischen Elemente, sowie menübasierte Parametrisierung der graphischen Attribute (Ebene 3).

⋈⋈⋈

C. Besonderheiten und Bewertung

Die erzeugten graphischen Elemente sind alleinstehende Bilder, solange es keine Möglichkeit gibt, sie in die Benutzungsschnittstelle einzubinden. Oft werden die Graphiken nur als Bilder, z. B. für den Hintergrund, oder als Aufschrift für einen Knopf verwendet. Um gezeichnete Elemente wiederverwendbar als Widget im Entwicklungssystem zu integrieren, müssen sie als Objekte (oder Funktionen) in die entsprechenden Bibliotheken eingebunden werden. Obwohl Zeichenprogramme weit verbreitet sind, ist diese Möglichkeit selten vorhanden. Beispielsweise bieten die Werkzeuge *grinx* oder *sphinx open* (eine Realisierung in Java) [in-00a, in-00b] solche Zeichenfunktionen an. Die Graphik kann auch auf der Benutzungsoberfläche plaziert, jedoch nicht als Widget, eingebunden werden.

∼∼∼

D. Realisierung im Werkzeug COMBO

Das Zeichenprogramm in COMBO ist für die Integration der erzeugten Graphik als Widget ausgelegt. Gezeichnete Elemente können als vordefinierte Elemente im Anordnungswerkzeug für die Benutzungsschnittstellen-Entwicklung zur Verfügung gestellt werden.

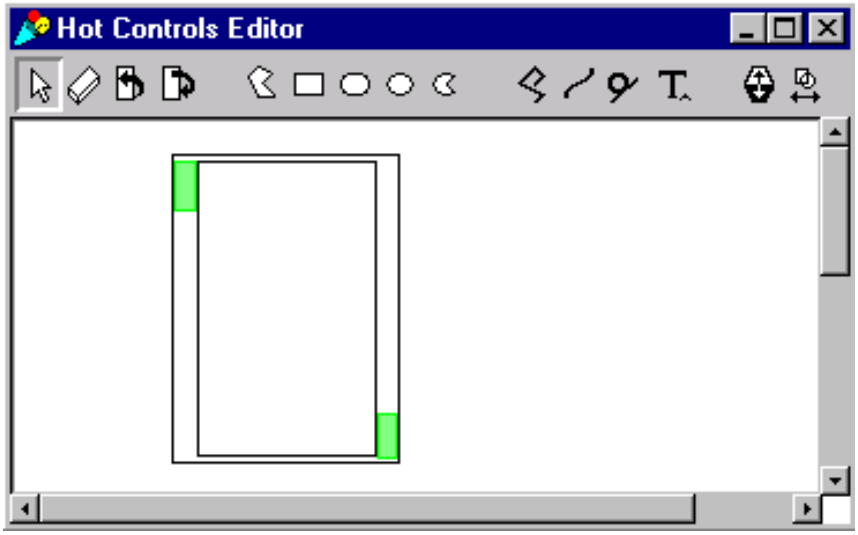


Abbildung 6.6: Erstellung des Tank-Widgets mit Hilfe des Zeichenprogramms

6.1.3 Layoutbibliothek

⋆ ⋆ ⋆

A. Einführung und Notation

Diese Methode spiegelt die Entwurfsmuster-Sprache „Anwendungsspezifisches Layout" aus Kapitel 4.4.2.2 wider. Sie ist eine Formatierungshilfe, die die Anwendung bestimmter Grundmuster im Layout für vordefinierte Anwendungsgebiete erlaubt. Die Auswahl erfolgt z. B. mit Hilfe einer Palette.

Die Layoutbibliothek hat, außer der Auswahl auf einer Palette, keine eigene Notation, sondern ist eher eine Art Hilfe-Funktion. Die Auswahl eines vordefinierten Layouts wird durch die Plazierung der entsprechenden Widgets auf der Arbeitsfläche interpretiert. Das Layout wird dann entweder mit Hilfe eines Zeichenprogramms oder eines Anordnungswerkzeugs definiert, je nachdem zu welchem Werkzeug die Zeichenfläche gehört, siehe Abbildung 6.7. Die Interpretation der Widgets verläuft dann wie bei den einzelnen Werkzeugen beschrieben.

⊙ ⊙ ⊙

B. Konzeptuelles Modell und Programmierebene

Die Layoutbibliothek basiert, wie die Anordnungswerkzeuge, auf der Vorstellung von der Verwendung vordefinierter Elemente und Strukturen, die die mentalen Anforderungen verringert. Die Benutzung der Layoutbibliothek findet, sofern sie durch direkte Manipulation auf einem Anordnungs- oder Zeichenwerkzeug angegeben werden kann, wieder auf der aufgabenorientierten Ebene (Ebene 3) statt.

⋈⋈⋈

C. Besonderheiten und Bewertung

Der Layoutbibliothek liegt die Entwurfsmuster-Sprache „Anwendungsspezifisches Layout" zugrunde. Es handelt sich also, wie bei Entwurfsmuster-Sprachen üblich, um ein baumstrukturiertes Vorgehen, das die einzelnen Elemente immer weiter verfeinern kann. Dadurch wird die Verwendung vordefinierter Layouts nicht zu statisch, und jedes Element kann, statt in die nächste Baumebene verfeinert zu werden, durch eigene Elemente ersetzt werden.

Bei der Realisierung unter Verwendung einer Palette, d. h. relativ kleiner Abbildungsflächen für die Auswahl, ist die Kenntnis der Entwurfsmuster-Sprache erforderlich. Durch die Verwendung von z. B. Assistenten könnte die Entwickler/in besser geführt werden.

~~~

E. Realisierung im Werkzeug COMBO

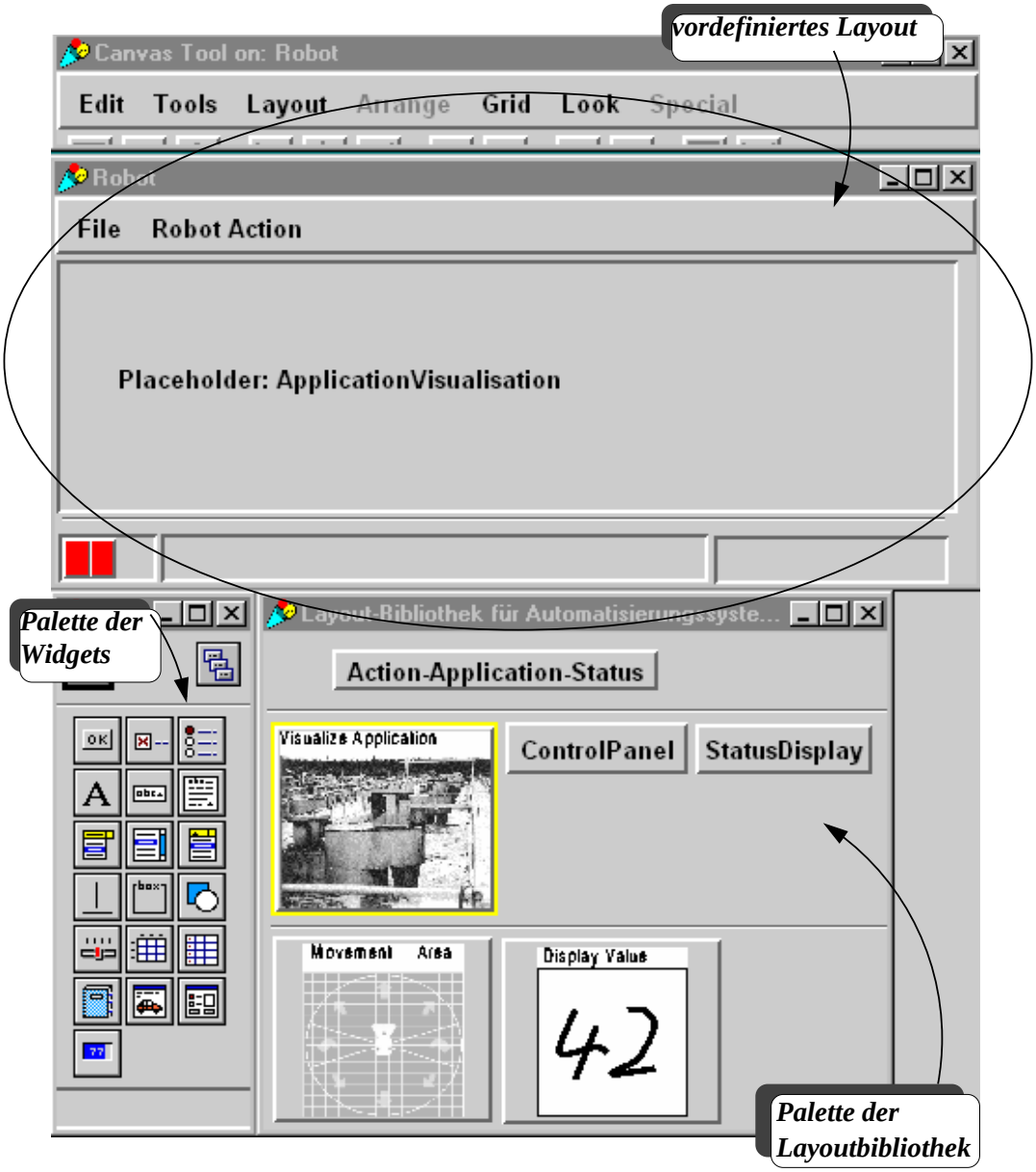


Abbildung 6.7: Die prototypische Implementierung der Layoutbibliothek

Die Auswahl der Layouts ist durch eine Palette vorgegeben. Die Auswahl „Action-Application-Status" erzeugt eine Layoutschablone, in der oben ein Menü, in der Mitte ein Platz für die Visualisierung der Anwendung und unten Platz für eine Statusleiste eingetragen wird. Solch ein Layout kann zunächst der Anwendung angepaßt werden, und die Platzhalter können mit weiteren Mustern verfeinert werden.

Die Abbildung der einzelnen Muster auf das Layout ist durch Smalltalk Klassen realisiert, z. B. wird das Muster *„VisualizeApplication"* durch einen `ViewHolder` implementiert.
~~~

6.2 Der *Strukturaspekt*: Elemente einer Benutzungsschnittstelle und ihre Struktur

Die „Baupläne" objektorientierter Systeme werden in der Regel durch **Klassen-** und **Objektdiagramme** dargestellt.

Dazu können prinzipiell alle Klassendarstellungen gängiger OOA/D-Methoden verwendet werden, z. B. die Unified Modeling Language (UML) [BRJ97, Oes97], die Object Modeling Technique (OMT) - eine der Methoden, die Grundlage der UML war [RBP+91] - oder OOA/D nach Coad und Yourdon [CNM95, CY91, CN94]. Eine Übersicht gibt z. B. [Sie92]. Im Rahmen dieser Arbeit soll die visuelle Darstellung dieses Aspekts der Entwickler/in die Struktur der Elemente des „hinter" der Oberfläche liegenden Programms zeigen, wie Baupläne, die die hinter der Fassade liegenden Elemente und Strukturen eines Hauses zeigen.

Alle Änderungen an anderen Aspekten, die Auswirkungen auf die Objekt- oder Klassenstruktur haben, sollen in den Strukturdiagrammen sichtbar werden. Schneider [Sch95] betont, daß die Darstellung solcher Strukturen eng an ein Anordnungswerkzeug angebunden werden sollte. Im Gegensatz dazu erfolgt bei den meisten OOA/D-Werkzeugen eine Trennung der beiden Darstellungen, indem die Klassen- und Objektstruktur in dem Werkzeug gezeigt wird, während die Entwicklung des Aussehens der Benutzungsschnittstelle in eine, in einer späteren Phase benutzten Programmierumgebung integriert ist.

Besonders in den Werkzeugen, die speziell zum Erstellen von Anwendungen in technischen Bereichen gedacht sind, wird die Struktur des Programms „hinter" der Benutzungsoberfläche mit Hilfe von Metaphern oder Analogien dargestellt:

- LabViews [Nat00], ein Werkzeug für die Meßtechnik, zeigt die Blockstruktrur der Anwendung.
- HP VEE [HP98], ebenfalls aus dem Bereich Meßtechnik, zeigt die einzelnen Oberflächenelemente als Blöcke, jedoch nicht in ihrer Anordnung auf der Oberfläche, sondern mit Beziehungen zwischen ihnen, die beispielsweise Signalaustausch bedeuten.
- Eine ähnliche Darstellung verwendet auch Simulink für die Regelungstechnik, ein Werkzeug, das in MatLab [Sci00] integriert ist.
- In der Darstellung in Electronics Workbench [Int96] können einzelne Schaltungselemente und die Schaltungen dargestellt werden.

Solche anwendungsspezifischen Darstellungen sind sehr gut, wenn sie ausschließlich in einem technischen Bereich angewendet werden. Meistens sind die Entwickler/innen damit an vordefinierte Blöcke oder Schaltungselemente gebunden, bzw. müssen andere Blöcke und Elemente von Hand programmieren. Der Code wird oft automatisch aus den Darstellungen generiert, daher ist es schwer, neue Programmierungs-Technologien, wie beispielsweise den Einsatz von Entwurfsmustern oder Komponenten-Technologie, zu integrieren.

Neben Klassen- und Objekt-Strukturen sind weitere Strukturen darstellbar: Beispielsweise können **Aufgabenmodelle** (d. h. die in der Analyse identifizierten Aufgaben der zukünftigen Benutzer/in) visuell dargestellt werden:

- Aufgaben werden im sog. Teallach Task-Modell [GPGW99] visuell durch Piktogramme, die die Art der Aufgabe beschreiben, innerhalb einer Baumstruktur dargestellt.

- Das Werkzeug MOBI-D [Pue99] bietet ebenfalls eine Darstellung der Aufgaben und ein Abbildungswerkzeug von Aufgaben zu Oberflächenelementen.

Da in der vorliegenden Arbeit die frühen Phasen der Entwicklung von Benutzungsschnittstellen nicht berücksichtigt werden, findet sich auch keine Umsetzung für Strukturen wie Aufgabenmodelle in COMBO.

6.2.1 Darstellung der Objektstruktur

⋆ ⋆ ⋆

A. Einführung und Notation

In der Darstellung der Objektstruktur werden alle Ausprägungen dargestellt, die für die Benutzungsschnittstelle erzeugt werden. Als Notation wird am besten eine der bekannten Objektdarstellungen gewählt, z. B. die UML.

⊙ ⊙ ⊙

B. Konzeptuelles Modell und Programmierebene

Die Darstellung der Objektstruktur als Bauplan soll die Entwickler/in dabei unterstützen, bewußt objektorientierte Programmierung einzusetzen. Jedes Einfügen eines Widgets erzeugt neue Objektelemente in der Objektvisualisierung und somit wird für die Entwickler/in die Beziehung zwischen objektorientiertem Programm und Anordnung der Benutzungsschnittstelle direkt sichtbar.

Es wird auch sichtbar, aus welchen Objekten sich komplexere Strukturen zusammensetzen, z. B. die anordnungsspezifischen Layouts (siehe Abschnitt 6.1.3), sofern beide Notationen parallel verwendet werden.

⋈⋈⋈

C. Besonderheiten und Bewertung

Oft wird übersehen, daß gerade Endbenutzer/innen Schwierigkeiten haben, objektorientiert zu programmieren, insbesondere, wenn es sich bei der Zielsprache um eine hybride Programmiersprache, wie C++, handelt. Eine besondere Schwierigkeit ist auch die Erzeugung von Ausprägungen in Abgrenzung zur Definition von Klassen. Beim textuellen objektorientierten Programmieren ergibt sich diese Schwierigkeit nicht, da beim Erlernen der Syntax genau diese Unterscheidung stattfindet.

Z. B. definiert „`Model subclass #myModel`" eine Klasse,
„`tmpModel := myModel new.`" erzeugt eine Ausprägung.

Beim Benutzen von Anordnungswerkzeugen verschwindet dieser Unterschied. Durch die Unterscheidung von Objekt- und Klassenstruktur wird genau dieser Schwierigkeit Rechnung getragen. Ob das der Entwickler/in wirklich hilft, ist allerdings noch nicht empirisch überprüft worden, da eine solche Überprüfung den Rahmen der Arbeit gesprengt hätte.

∼∼∼

D. Implementierung im Werkzeug COMBO

Es wird eine UML-ähnliche Notation verwendet, in der Objekte als Rechtecke mit ihren Attributen und Methodennamen dargestellt werden.

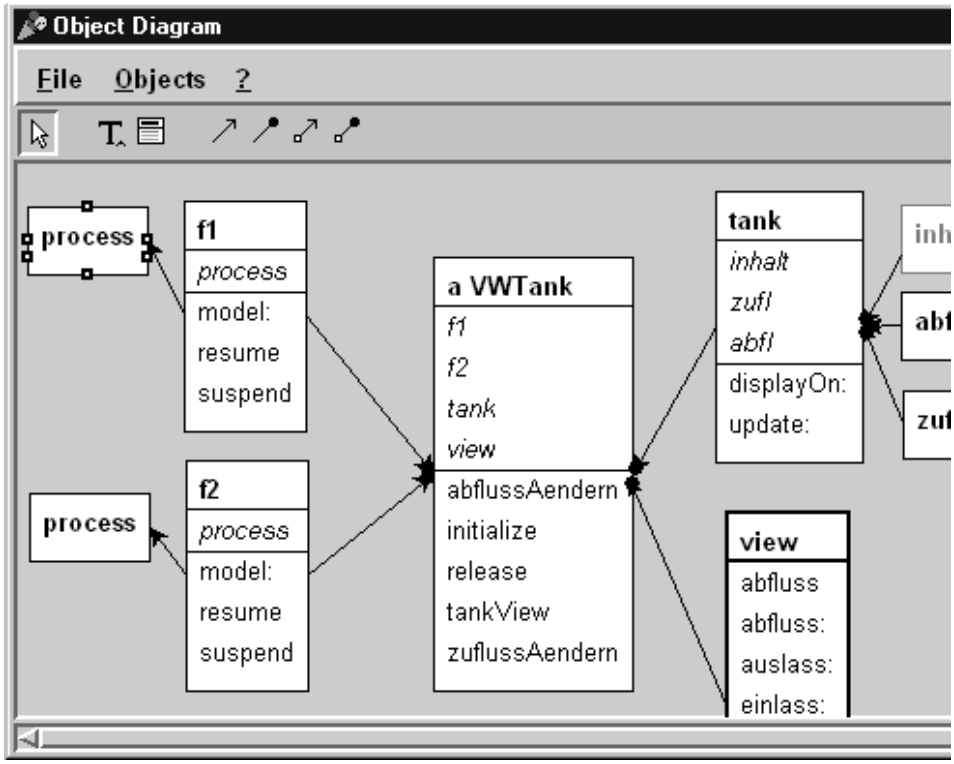


Abbildung 6.8: Darstellung der Objektstruktur im Tankbeispiel

6.2.2 Klassendiagramme

⋆ ⋆ ⋆

A. Einführung

Klassendiagramme sind in vielen OOA/D-Werkzeugen ebenfalls weit verbreitet. Es ist gut, eine bekannte Notation zu wählen, insbesondere wenn neben COMBO auch OOA/D- oder andere Werkzeuge verwendet werden.

⊙ ⊙ ⊙

B. Konzeptuelles Modell und Programmierebene

Das konzeptuelle Modell der Benutzung von Klassendiagrammen wurde am Anfang dieses Abschnitts bereits beschrieben: Klassendiagramme stellen Baupläne objektorientierter Systeme dar. Es handelt sich hier ebenfalls um eine visuelle Darstellung der Struktur (Ebene 2).

⋈⋈⋈

C. Besonderheiten und Bewertung

Auch Klassendiagramme helfen der Entwickler/in bei der gedanklichen Umsetzung der Elemente der Benutzungsschnittstelle durch objektorientierte Programmierung. Falls es sich um eine Endbenutzer/in handelt, die nicht mit der objektorientierten Programmierung vertraut ist, helfen die Klassendiagramme beim „objektorientierten Denken": Indem deutlich wird, welche Beziehungen zwischen Klassen existieren, lernt die Entwickler/in den Aufbau einer Benutzungsschnittstelle anhand der „Beispiele", die sie selbst beim Entwurf erzeugt.

$\sim\sim\sim$

D. Implementierung im Werkzeug COMBO

Auch in COMBO wird eine UML-ähnliche Notation verwendet, allerdings werden für die Erbungsbeziehung OMT-Symbole [RBP+91] verwendet.

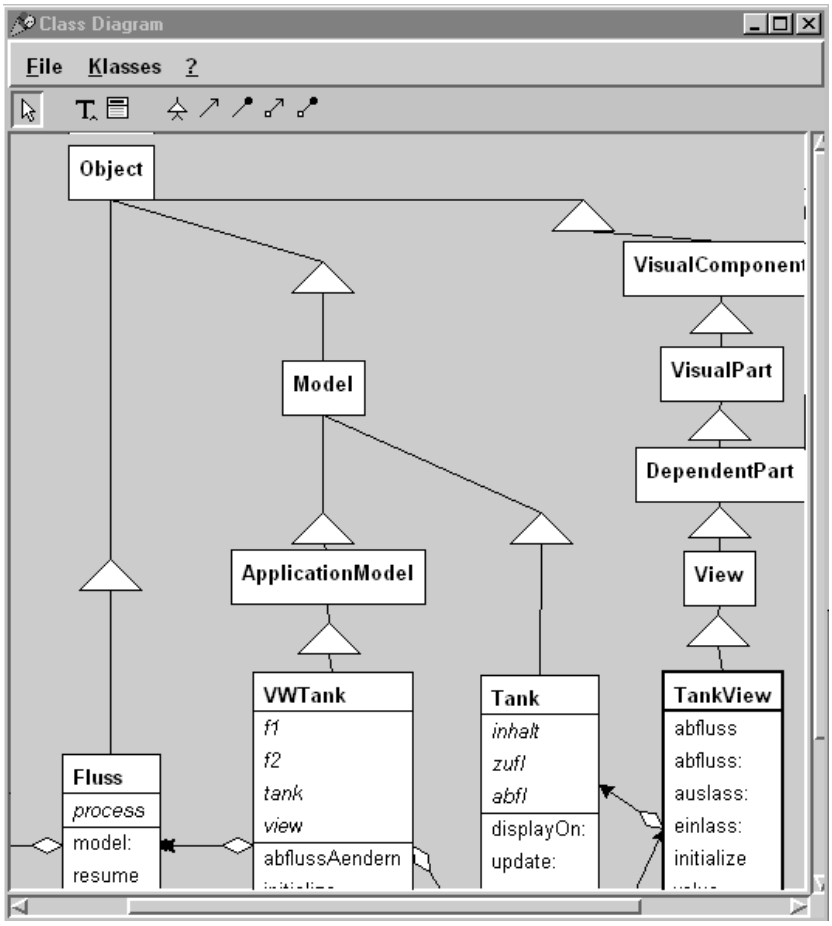


Abbildung 6.9: Klassenstruktur des Tankbeispiels

6.3 Der *Verhaltensaspekt*: das Verhalten der Benutzungsschnittstelle

In den letzten zwei Abschnitten wurden zwei grundlegende Aspekte von Benutzungsschnittstellen beschrieben, nämlich das Aussehen und das objektorientierte Programm „dahinter".

Nun soll den Elementen in den beiden Sichten Leben eingehaucht werden, d. h. ihr ***Verhalten*** soll definiert werden. Dieses Verhalten ist der dritte grundlegende Aspekt der Benutzungsschnittstelle.

Wie in Abbildung 6.10 dargestellt, ist die Benutzungsschnittstelle ein objektorientiertes System, das (bis hierher) zum einen durch den Bauplan, d. h. die Darstellung der Klassen/Objekte visualisiert wird, und zum anderen durch seine „Fassade", d. h. die Anordnung sichtbarer Elemente. Aufbauend auf diesem Verständnis wird „das Verhalten" definiert: Wie

in Kapitel 5 beschrieben, wird es aus den Teilaspekten *graphisches Verhalten*, *Objektverhalten* und *Verhalten durch Mechanismen der Klassenbibliothek* zusammengesetzt, die in den folgenden Abschnitten beschrieben werden.

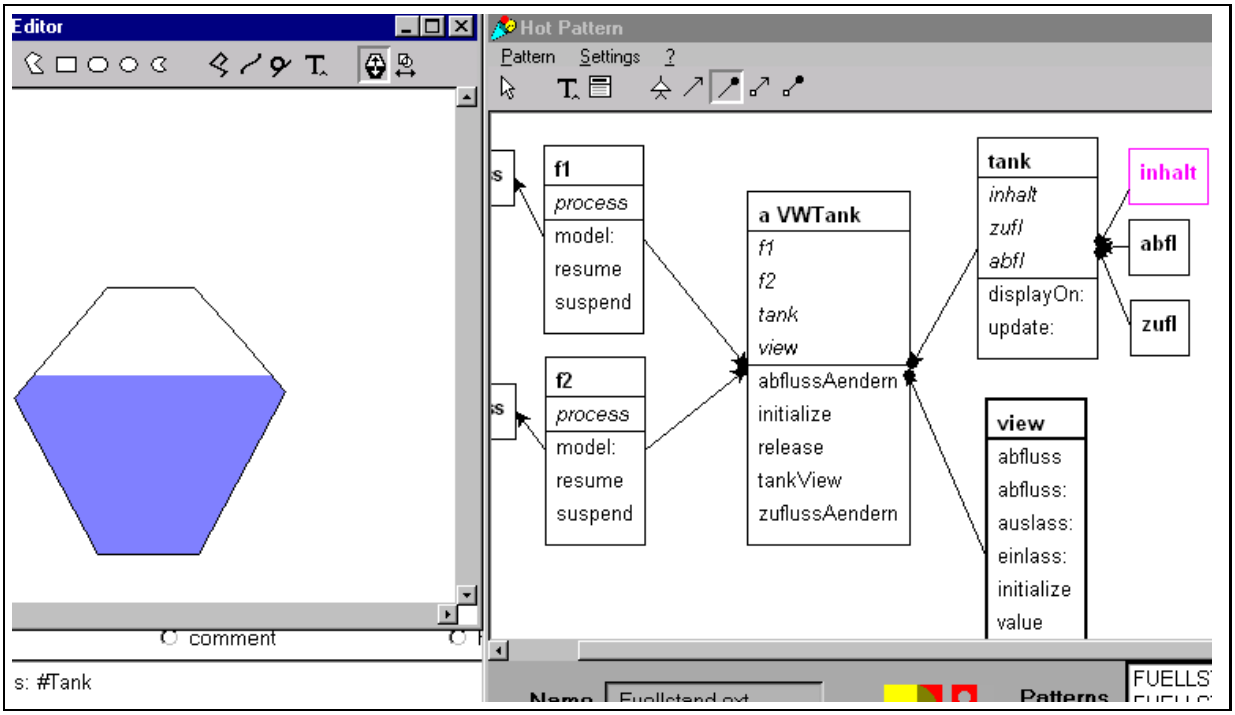


Abbildung 6.10: Zwei Aspekte des Tankbeispiels: Darstellung der Anordnung (der Tank ist hier in einem erweiterten Beispiel als Sechseck dargestellt) und der Objektstruktur

Definition von Verhalten durch visuelle Programmiersprachen

Die meisten visuellen Programmiersprachen wurden entwickelt, um das Verhalten von Software-Systemen zu beschreiben. Solche Programmiersprachen sind oft als Allzweck-Programmiersprachen erfunden. Eine vollständige Auflistung aller verfügbaren Sprachen würde den Rahmen dieser Arbeit sprengen. Beispielhaft seien hier folgende Sprachen zur Verhaltensdefinition teilweise (in Abschnitt 6.2) bereits eingeführter Systeme genannt:

- **LabViews** [Nat00]: Enthält neben dem Anordnungswerkzeug auch eine prozedurale visuelle Sprache.
- **Prograph** [SC95, CS96] ebenso wie **VisaVis** [Pos96]: Visuelle funktionale Allzwecksprachen, mit unterschiedlichen Visualisierungen.
- **VisPro** [Zha98]: Eine regelbasierte Sprache zur Definition von Verhalten graphischer Objekte. Auch **ChemTrains** [BL93] definiert das Verhalten durch eine regelbasierte visuelle Sprache. Sowohl die Regeln als auch die Aktionen werden visuell dargestellt.
- **Agentsheets** [RC93, Zha98]: Das Verhalten von graphischen Elementen, die als Agenten verstanden werden, wird durch Zustandsmodelle der einzelnen Agenten spezifiziert. Die Kommunikation der Agenten erfolgt durch Schließen aufgrund ihrer räumlichen Anordnung.
- **Fabrik** [LCI+88]: Das graphische Verhalten wird, ähnlich einer Schaltung, aus Datenquellen und Verarbeitungsblöcken zusammengesetzt.

- **Pictorial Janus** [Kah92]: Das Verhalten wird durch die Kommunikation spezialisierter animierter Visualisierungen von Objekten, Datenstrukturen, Agenten oder Funktionen beschrieben.
- **Form/Formula** [KASC95] und **Forms/3** [BCA00, GBK+00]: Wie bei einer Tabellenverarbeitung wird der Zusammenhang zwischen Zellen mit einer visuellen Programmiersprache definiert.
- **Vista** [Sch97a]: Verwendet die Metapher von Chips zur Programmierung von Oberfläche und Verhalten.

Warum werden dann noch weitere visuelle Spezifikationsmethoden für das Verhalten benötigt?

- Die im Rahmen dieser Arbeit vorgenommene Aufteilung in Aspekte erlaubt die Benutzung *einfacher* visueller Methoden, denn mit jeder visuellen Methode muß nur dieser Aspekt definiert werden.
- Es können *spezialisierte* visuelle Methoden verwendet werden, die
  - genau an die Erfordernisse eines Aspekts angepaßt sind oder
  - speziell an die Anforderungen von Benutzungsschnittstellen angepaßt sind

Übersicht über visuelle Methoden für den Teilaspekt *graphisches Verhalten*

Zum graphischen Verhalten gehört alles, was auf der Benutzungsoberfläche sichtbar ist, z. B.

- Änderungen der Attribute graphischer Elemente (z. B. Farben, Größe, Position) in Abhängigkeit
  - von den Attributen anderer graphischer Elemente,
  - von Daten, die die Anwendung liefert und
  - von der Zeit.
- Eingabeverhalten

Oft ist das Verhalten bei vordefinierten Widgets schon ausprogrammiert und wird damit durch die Anordnung „mitdefiniert". Das hat einerseits den Nachteil, daß eine Verhaltensänderung die Änderung auf Codeebene bedeutet, und andererseits sind die Aspekte für die Entwickler/in nicht klar getrennt. Diese Trennung ist bei der ausschließlichen Verwendung vordefinierter Elemente nicht erforderlich, für die Entwicklung neuer Widgets oder die Änderung vorhandener Widgets jedoch notwendig.
Für die visuelle Definition graphischen Verhaltens gibt es viele Ansätze:

- **Programmieren durch Vormachen oder durch Beispiele**, z. B. im Werkzeug Peridot [Mye88] oder in den in Abschnitt 6.1 beschriebenen Systemen.

- Programmieren durch **graphische Constraints** [Szw92], wie in Abschnitt 6.1 bereits beschrieben. Constraints sind zur Spezifikation des graphischen Verhaltens besonders gut geeignet, zur Definition der Anordnung sind sie eigentlich zu mächtig. Dabei können die Constraints selbst visuell dargestellt werden [Hül97], oder das Constraint-Netz, siehe z. B. [Pal95, Bor78, Bor86]. Da COMBO jedoch keine Constraints benutzt und auch keinen Constraintlöser (siehe Abschnitt 6.1) besitzt, sind diese visuellen Programmiermöglichkeiten nicht weiter untersucht worden. Für eine Erweiterung von COMBO könnten solche visuellen Sprachen jedoch interessant sein.

- Im Bereich des graphischen Verhaltens ist der Einsatz von **Entwurfsmuster**n interessant, z. B. eine Umsetzung der in Abschnitt 4.4.2 beschriebenen Entwurfsmustersprache für graphisches Verhalten. Solch eine Realisierung wird unter dem Namen *Verhaltensbibliothek* im nächsten Abschnitt (6.3.1) vorgestellt.

- **Visuelle Programmiersprachen** für graphisches Verhalten, wie in der Einführung zu diesem Abschnitt beschrieben.

6.3.1 Verhaltensbibliothek

⋆ ⋆ ⋆

A. Einführung und Notation

Die Verhaltensbibliothek setzt die Entwurfsmustersprache für Oberflächenverhalten aus Abschnitt 4.4.2.1 um. Sie ist der Layoutbibliothek aus Abschnitt 6.1.3 ähnlich, indem auch hier vordefinierte Teilaspekte verwendet werden, in diesem Fall vordefiniertes Verhalten. Die Idee dabei ist, daß einem graphischen Element ein Verhalten zugeordnet wird, das vorher aus einer Palette von möglichen Verhalten ausgewählt wurde. Die Interpretation ist, daß in die Klasse, die dieses graphische Element beschreibt, alle Attribute und Methoden als Code hinzugefügt werden, die es braucht, um das Verhalten zu implementieren. Als eigene Notation sind neben der Definition der Palette Anmerkungen denkbar, die an die graphischen Elemente angebracht werden.

Diese visuelle Spezifikationsmethode beruht auf dem Prinzip „Auswählen und Beschreiben", das in Kapitel 2 vorgestellt wurde, durch zweidimensionale Auswahltechniken und Beschreibung mit Parameter.

⊙ ⊙ ⊙

B. Konzeptuelles Modell und Programmierebene

Das konzeptuelle Modell bei der Benutzung der Verhaltensbibliothek entspricht der Vorstellung, daß graphische Elemente ein Verhalten bekommen sollen. Auch Dokumentenbearbeitungswerkzeuge, z. B. PowerPoint von Microsoft [Mic00a], benutzen diese Vorstellung. In PowerPoint z. B. kann einem Element einer Folie das Verhalten „erscheine bei Tastendruck von links auf der Folie," zugeordnet werden.

Die Definition von Verhalten ist unmittelbar und damit auf der aufgabenorientierten Ebene (Ebene 3).

⋈⋈⋈

C. Besonderheiten und Bewertung

Die Verhaltensbibliothek wird von den Entwickler/innen als besonders einfach und intuitiv empfunden (siehe z. B. Abschnitt 7.4).

∼∼∼

D. Realisierung im Werkzeug COMBO

Im Gegensatz zur Layoutbibliothek wurde die Verhaltensbibliothek nicht explizit als Palette realisiert, sondern im Zeichenwerkzeug in der Leiste für die verschiedenen Zeichenmodi untergebracht.

Zur Demonstration sind die Muster „moving linear" und „be filled" implementiert worden.

Die folgenden Abbildungen 6.11 bis 6.13 zeigen das Hinzufügen des Musters „be filled" zu einem Tankelement.

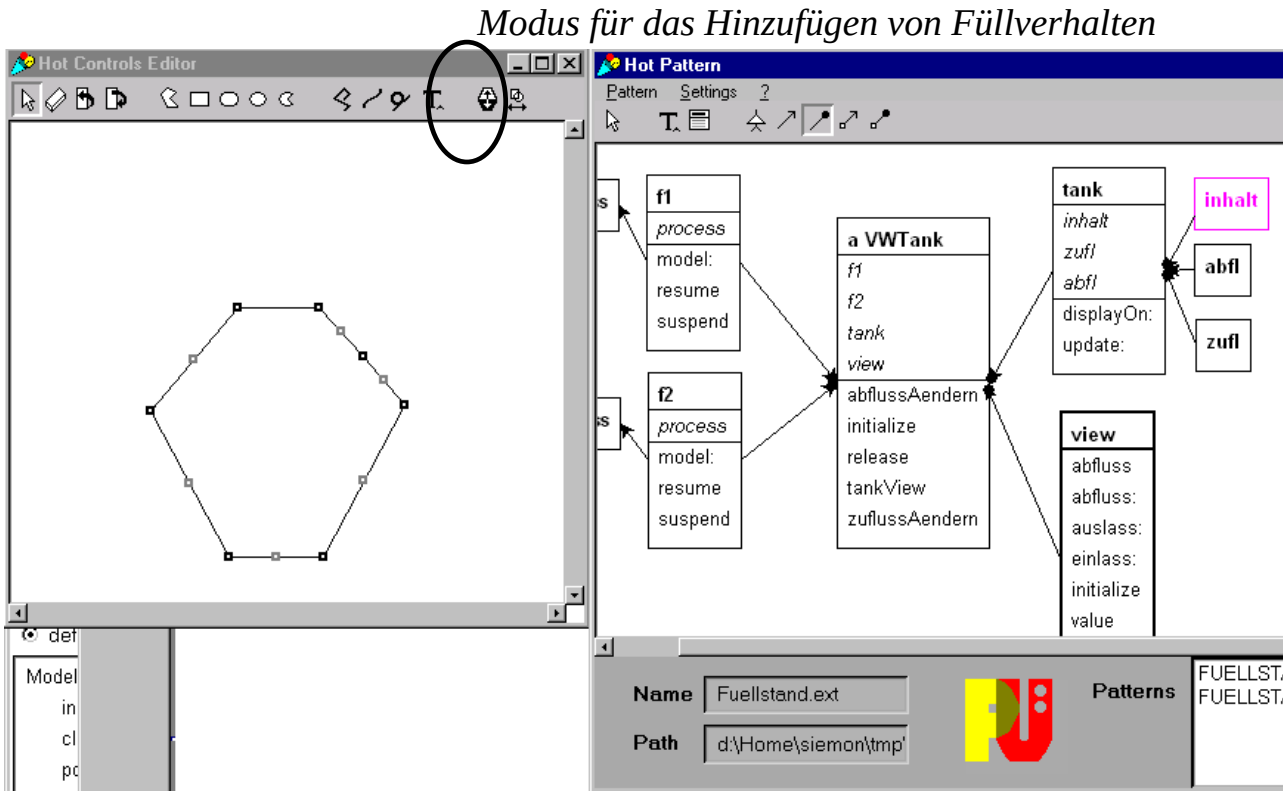


Abbildung 6.11: Schritt 1 - Auswahl des Modus zum Hinzufügen von Füllverhalten

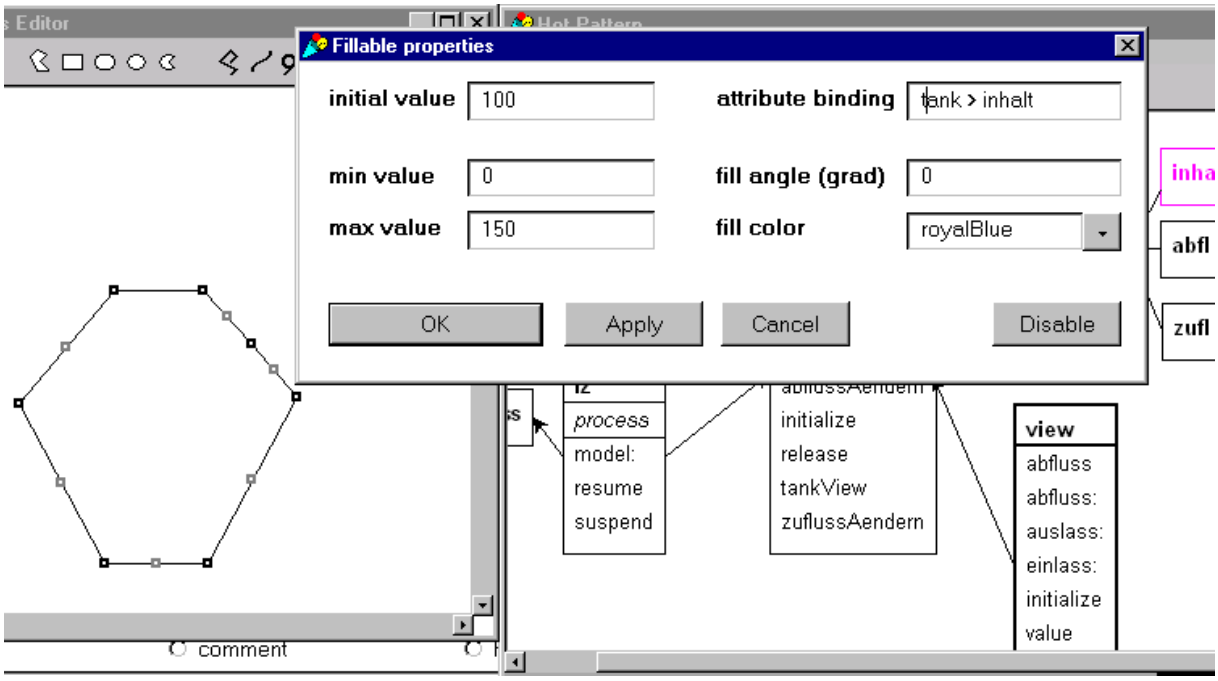


Abbildung 6.12: Schritt 2 - Auswahl des Modus zum Hinzufügen von Füllverhalten: Definition der Parameter. Der Tank wird in Abhängigkeit vom Attribut `inhalt` des Objekts Tank gefüllt

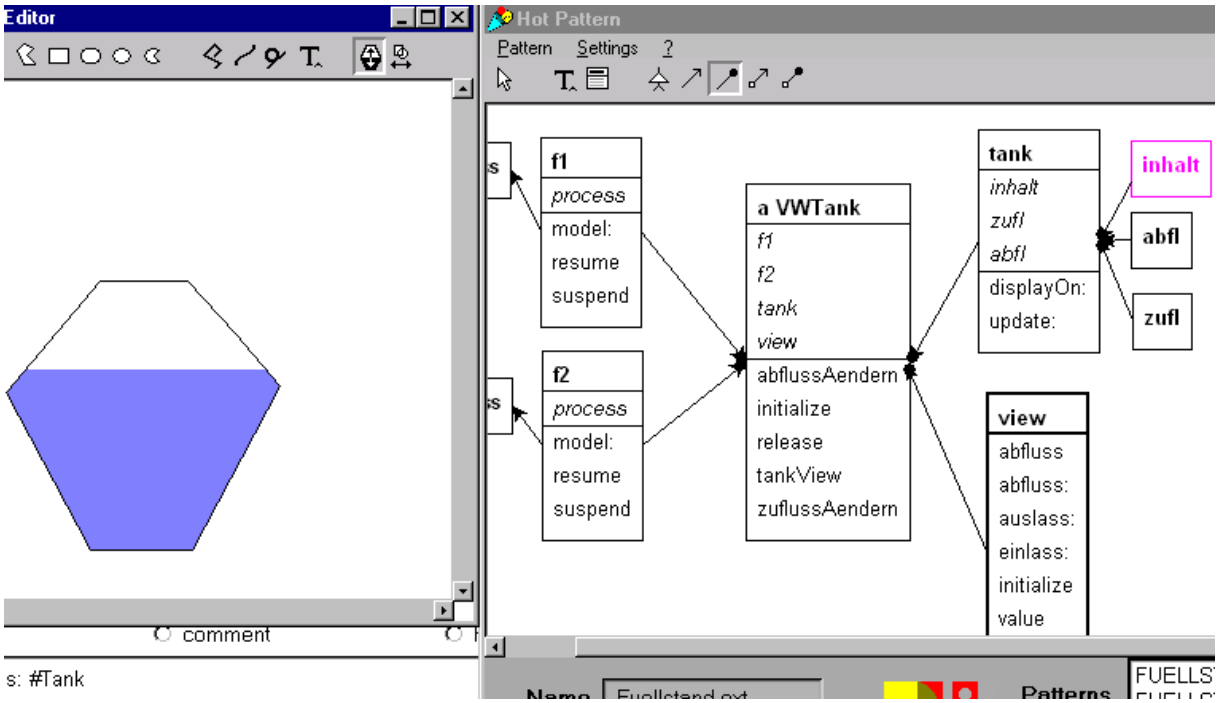


Abbildung 6.13: Schritt 3 - Ausprobieren des Füllverhaltens: Zuordnen des Füllverhaltens zu einem Tankelement

Übersicht über visuelle Spezifikationsmethoden für den Teilaspekt *Objektverhalten*

Es gibt viele Ansätze, das Verhalten objektorientierter Systeme auf der Objektebene mit visuellen Notationen zu beschreiben. Die Schwierigkeit liegt darin, daß Objekte und Klassen unterschieden werden müssen und das Verhalten auf mehreren Granularitätsstufen beschrieben werden kann: Möglich ist die Beschreibung des Verhaltens

- der Benutzungsschnittstelle als Ganzes bzw. einer Anwendung,
- einer Gruppe von Objekten (z. B. Entwurfsmuster oder Ausschnitte der Klassenbibliothek),
- eines Objekts,
- einer Methode eines Objekts.

Schon die UML bietet - ohne die Notation für Entwurfsmuster und Komponenten - fünf verschiedene Notationen für das Verhalten eines objektorientierten Systems an.

Zustands-Übergangsdiagramme und **Datenflußdiagramme** sind weit verbreitet und werden als bekannt angenommen. **Sequenzdiagramme** [BRJ97] zeigen den zeitlichen Ablauf von Methodenaufrufen zwischen mehreren Objekten. **Anwendungsfalldiagramme** (**use case**-Diagramme) [Jac92] beschreiben Kommunikationssequenzen für einzelne Anwendungsfälle. Eine Übersicht findet sich z. B. in [Sie92].

Petrinetze sind eine graphische und formale Methode zum Beschreiben von Abläufen in dynamischen Systemen (siehe z. B. [Bau90, LE90, Rei85]). Sie werden als Graphen dargestellt, deren Knoten *Stellen* und deren Kanten *Transitionen* genannt werden. Die Stellen können mit sog. *Tokens* belegt werden, die Wertemengen repräsentieren.

Beim Übergang von einem Knoten zu einem anderen werden diese Tokens „verarbeitet", d. h. es werden welche verbraucht oder es kommen welche hinzu oder sie verändern sich. Übergänge können von den Belegungen abhängen. Der Zustand des jeweiligen Netzes ist durch die Markierung des Netzes mit Tokens repräsentiert.

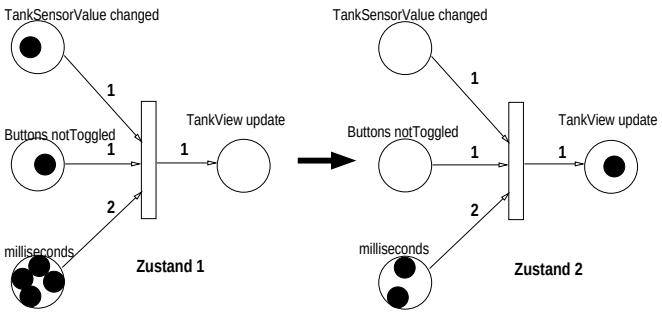


Abbildung 6.14: Zwei Zustände eines Petrinetzes

Wie in Tabelle 6.1 zu sehen ist, können die Petrinetze auf vielen Granularitätsstufen zur Definition des Verhaltens eines objektorientierten Systems eingesetzt werden. Im Bereich der Entwicklung von Benutzunggschnittstellen werden sie meistens eingesetzt, um Dialogabläufe zu spezifizieren. Es gibt viele Ansätze, die hier nicht einzeln angesprochen werden können.

Ein Problem ist bei den meisten Ansätzen, daß sie sowohl Klassen- als auch Objektverhalten abdecken wollen und daher neben der Klassenbeschreibung auch die Ausprägung und damit die Erzeugung der Objekte darstellen müssen. Dadurch werden die Netze oft unübersichtlich. In dieser Arbeit sollte daher nur ein Teil des Verhaltens durch Petrinetze dargestellt werden, in der Hoffnung, daß die Diagramme dadurch vereinfacht werden. Ein solcher Ansatz wird im nächsten Abschnitt dargestellt.

Eine weitere Methode, objektorientiertes Verhalten zu visualisieren, wurde von Koschorek [Kos94] entwickelt: der **Visual Method Browser**. Der Schwerpunkt liegt dabei auf der Visualisierung von Codesequenzen innerhalb einer Methode durch Verdeutlichung des Botschaftenaustauschs.

Tabelle 6.1 zeigt eine (nicht vollständige) Übersicht, welche Notationen typischerweise auf den verschiedenen Granularitätsstufen verwendet werden.

Granularitäts-stufe	**Notation für Objekte/Klassen**	**Notation unabhängig von Klassen und Objekten**
Benutzungs-schnittstelle	Petrinetze (PNO [BP90, BP95] PUIST [LMH97a, LMH97b])	Petrinetze (VIP [Haa87], Dialognetze (z. B. Genius [JWZ93]), SMN [Vie89], [BK93])
	Zustandsdiagramme	Zustandsdiagramme [ES95]
		Anwendungsfalldiagramme (Use Case-Diagramme) (UML [BRJ97])
Klassen/Objekt-Gruppen	Klassendiagramm (UML [BRJ97]), evtl. durch Assoziationen	Komponenten (UML)
	Eigenschaftsfenster für Widgets (VisualWorks [Par95])	
	Kollaborationsdiagramme (UML)	
	Petrinetze(Vis-A-vis [LS93], OPN)	
	Kommunikationsdiagramme (SM [SM92])	
Klasse/Objekt	Aktivitätsdiagramme (UML)	Petrinetze (HCPNS [Lak95])
	Kontrollablaufdiagramme (SM [SM92])	
	Datenflußdiagramme (OMT [RBP+91])	
Methoden	Aktions-Datenfluß-Diagramme (SM [SM92])	Petrinetze [Koc94, FM93, Sch92]
	Visual Method Browser [Kos94]	

Tabelle 6.1: Visuelle Notationen zur Beschreibung ojektorientierter Systeme

In Klammern die Notation der Vorgehensmethode oder des Systems:

UML	=	Unified Modeling Language	HCPNS	=	High Level Colored Petri Nets
SM	=	Shlaer-Mellor	PNO	=	Petri Net Objects
OMT	=	Object Modeling Technique	SMN	=	Strukturierte, Markierte Netze
PUIST	=	Petri Net based User Interface Specification Tool	OPN	=	Objekt-Petrinetze

6.3.2 Objekt-Petrinetze

⋆ ⋆ ⋆

A. Einführung und Notation

In Anlehnung an die Arbeiten von Becker und Moldt [BM93], das Werkzeug INCOME [PRO92] sowie andere Arbeiten (siehe z. B. [Sie92]), wurde im Rahmen dieser Arbeit eine Spezifiktionsmethode mit Petrinetzen entwickelt, mit der das Objektverhalten in Form von *Methodenaufrufen* zwischen den Objekten spezifiziert werden. Gegenüber den Sequenz- und Zustandsdiagrammen der UML haben Petrinetze den Vorteil, daß sie den Gesamtzustand des Systems zeigen.

Jedes Objekt wird als Teilnetz dargestellt und mit einem Kasten umrandet (Notation siehe unten). Methoden und Attribute eines Objekts werden durch eine Kombination aus Stelle-Transition-Stelle dargestellt.

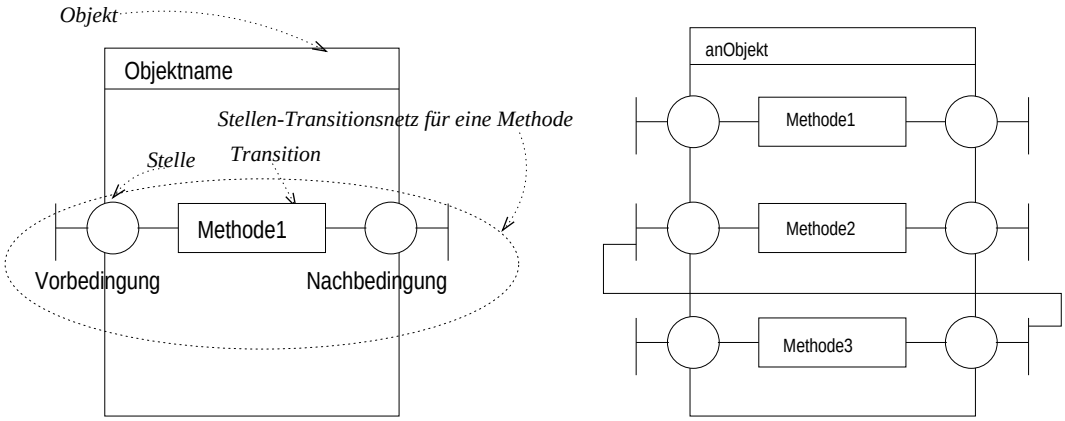


Abbildung 6.15: Notation für Objekt-Petrinetze und ein Objektnetz

Sobald in einer Stelle ein Token liegt, wird die dazugehörige Methode aufgerufen, d. h. durch solche Netze werden Methodenaufrufe von Objekten visualisiert.

⊙ ⊙ ⊙

B. Konzeptuelles Modell und Programmierebene

Neben dem Festlegen der inneren Zustände eines Objekts wird das Verhalten objektorientierter Systeme im wesentlichen durch die Kommunikation zwischen den Objekten, d. h. die gegenseitigen Aufrufe von Methoden, definiert. Die Vorstellung ist daher, daß der aktuelle Zustand des Gesamtsystems durch die Darstellung der Methodenaufrufe (also eine Visualisierung auf der Strukturebene, Ebene 2) angezeigt wird. Durch Anordnungs- und Strukturaspekt wurde festgelegt, aus welchen Objekten die Benutzungsschnittstelle besteht, so daß die Objekte als Objektnetze, wie in **A.** dargestellt, gezeichnet werden können. Die Abfolge der Methodenaufrufe legt dann das Verhalten fest. Umgekehrt wird die Kommunikation in der laufenden Benutzungsschnittstelle durch „wandernde" Tokens visuell dargestellt.

⋈⋈⋈

C. Besonderheiten und Bewertung

Diese Darstellung hat verschiedene interessante Aspekte: Das Verhalten der Benutzungsschnittstelle wird zur Laufzeit animiert; gleichzeitig kann es zur Entwicklungszeit definiert werden.

Die Notation ist sehr einfach, so daß die Darstellung übersichtlich ist.

Trotzdem wird die Darstellung mit wachsender Komplexität problematisch. Je mehr Methoden ein Objekt hat, desto größer wird es und nimmt sehr viel Platz weg. Außerdem stellt sich die Frage, welche Methoden dargestellt werden: Die Methoden, die aktuell im Rahmen der Entwicklung der Benutzungsschnittstelle definiert wurden oder auch alle ererbten Methoden? Ein weiteres Problem ist die Benutzung von Entwurfsmustern: Sollen z. B. Adapter-Objekte ebenfalls dargestellt werden?

~~~

D. Realisierung im Werkzeug COMBO

Die unter **C.** beschriebene Problematik hat zu der Entscheidung geführt, nur die von der Entwickler/in definierten Methoden und Objekte darzustellen.

Im Beispiel wurde die Tank-Benutzungsschnittstelle entsprechend den vorhergehenden Abschnitten folgendermaßen modelliert:

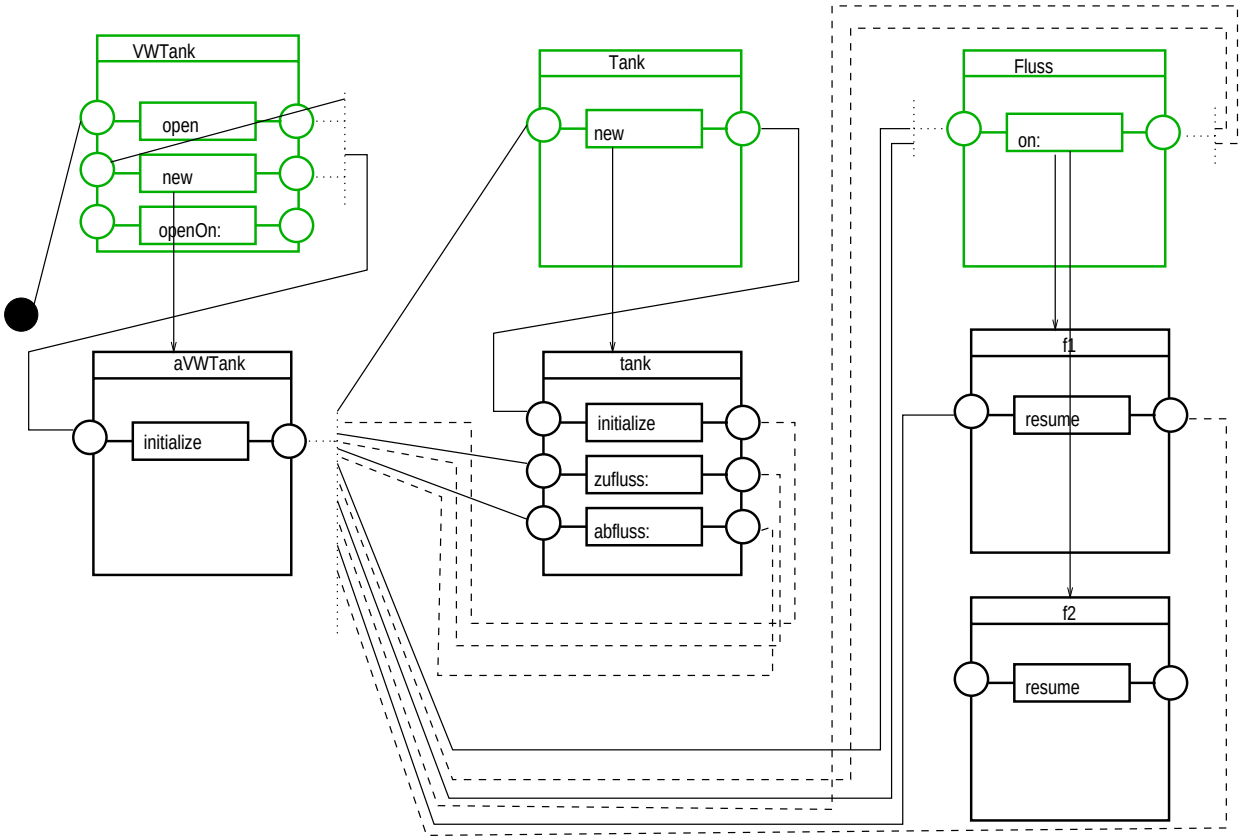


Abbildung 6.16: Das Objekt-Petrinetz für die Tank-Benutzungsschnittstelle

Das Vorgehen im einzelnen und die genaue Darstellung im Werkzeug COMBO werden in der von mir betreuten Studienarbeit von Hartmann [Har01] nachzulesen sein, beim Abschluß dieser Arbeit war sie noch nicht fertiggestellt.
~~~

6.3.3 Visual Method Browser

⋆⋆⋆

A. Einführung und Notation

Koschorek [Kos94] hat diese visuelle Darstellung von einzelnen Methoden entwickelt. Sie wurde im Rahmen einer von Koschorek betreuten Diplomarbeit [Gud95] implementiert. Objekte werden durch Rechtecke dargestellt, der Aufruf einer Methode dieses Objekts wird durch einen Pfeil symbolisiert, wie in der folgenden Abbildung zu erkennen ist:

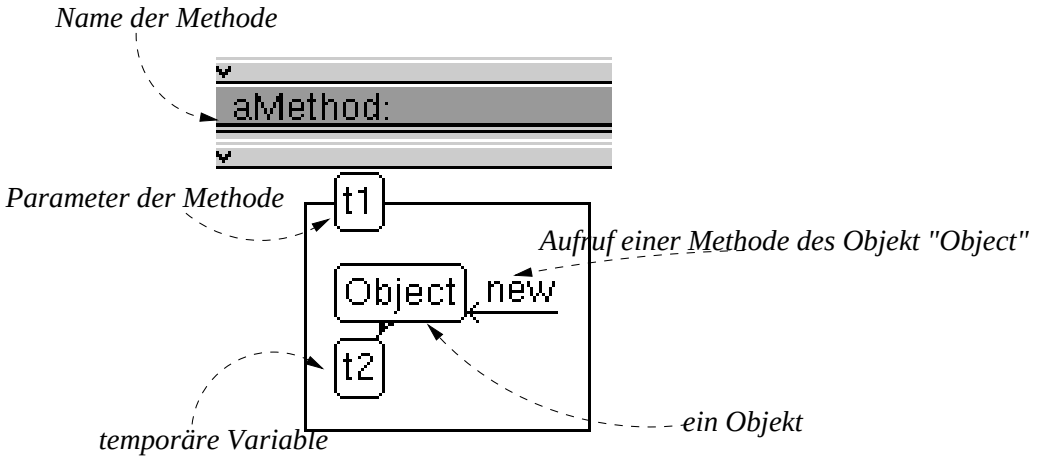


Abbildung 6.17: Notation des Visual Method Browsers

⊙⊙⊙

B. Konzeptuelles Modell und Programmierebene

Diese Darstellung arbeitet auf der Ebene von Basismechanismen (Ebene 0). Die Vorstellung ist, Variablen, d. h. Objekte, visuell darzustellen und den Moment ihrer Beteiligung zusammen mit der Methode zu verdeutlichen, die aufgerufen wird.

⋈⋈⋈

C. Besonderheiten und Bewertung

Durch die zweidimensionale Anordnung werden die Elemente (Objekte, temporäre Variablen, Parameter, Ergebnisse von Anweisungen) übersichtlich angeordnet. Das Ausnutzen der zwei Dimensionen entspricht beim Aufbau der Visualisierung den Anweisungen der textuellen Programmierung. Zusätzlich werden die Ergebnisse von Anweisungen abgehoben dargestellt und Verschachtelungen verdeutlicht.

Beim Arbeiten mit der Methode hat sich gezeigt, daß die Darstellung eher mehr Platz braucht als die textuelle Programmierung, daß sie in vielen Fällen aber leichter zu lesen ist, als der Text.

∼∼∼

D. Realisierung im Werkzeug COMBO

Die Implementierung der Methode konnte in COMBO direkt übernommen werden (bis auf die Unverträglichkeiten durch die verschiedenen Versionen der Entwicklungsumgebung), ist also keine Eigenentwicklung.

Im folgenden sollen einige Methoden des Tankbeispiels dargestellt werden:

Die Visualisierung der Methode `abflussAendern` des Objekts `VWTank` im Tankbeispiel sieht folgendermaßen aus:

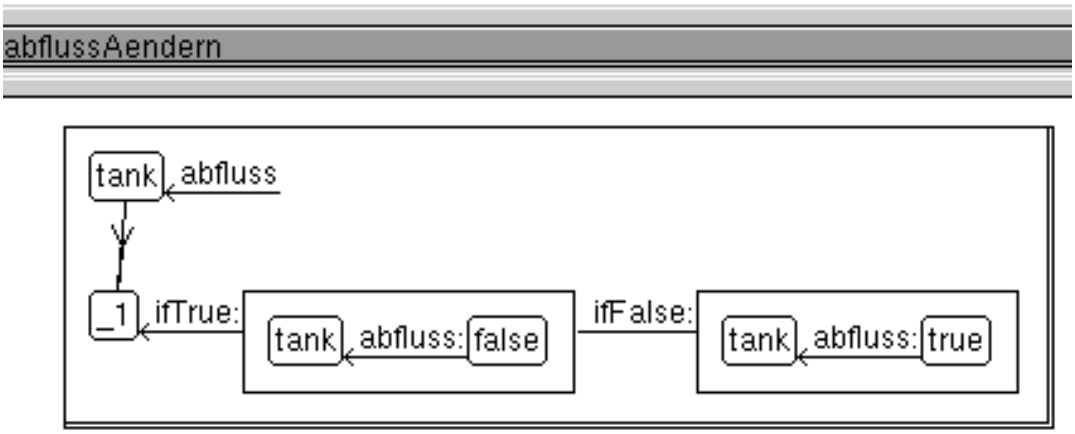


Abbildung 6.18: Die Methode `abflussAendern`

In diesem Beispiel werden die Blöcke sehr deutlich.

Durch eine solche Visualisierung werden auch Ähnlichkeiten klar erkennbar, was beim Wiederverwenden oder beim Finden von Mustern eine Rolle spielt. Ein Beispiel sind die sehr ähnlichen Methoden zur Modifikation des Tankinhalts der Klasse `Tank, einlass:` und `auslass:`

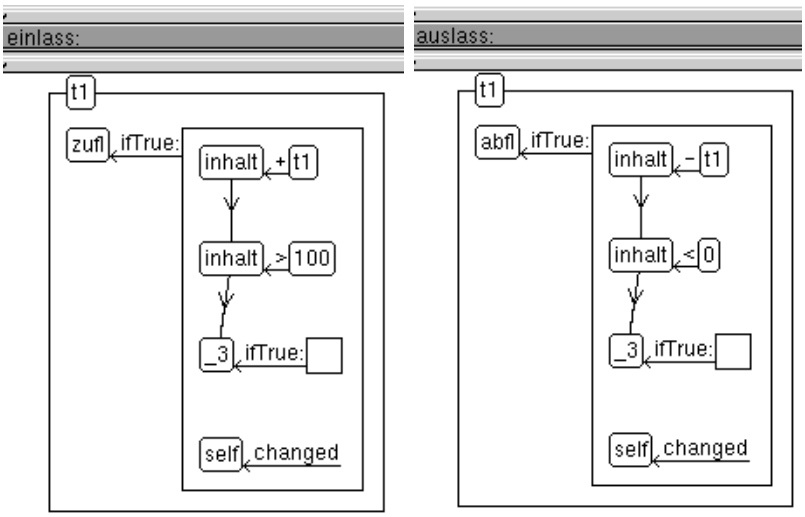


Abbildung 6.19: Die Methoden `einlass:` und `auslass:`

Übersicht über visuelle Methoden für den Teilaspekt *Verhalten durch Mechanismen und Muster der Klassenbibliotheken*

Stellvertretend für viele andere Mechanismen der Klassenbibliothek soll der MVC-Mechanismus visualisiert werden. Die Visualisierung solcher Mechanismen wird in den meisten Werkzeugen und Analyse- oder Designmethoden nicht explizit berücksichtigt. Statt dessen werden visuelle Notationen für Verhalten angeboten, mit denen sich solche Mechanismen ausdrücken lassen. Die Sequenzdiagramme der UML lassen sich z. B. für solche Visualisierungen benutzen[Lar99, Sel99] und bieten somit eine einheitliche Darstellung für alle Muster und damit ein einheitliches Verständnis für die Entwickler/innen an. Die Lücke zwischen aufgabenbezogener Modellierung (Ebene 3) und der Modellierung mit Basismechanismen (Ebene 1) wird dadurch aber nicht geschlossen

6.3.4 Visualisierung des MVC-Mechanismus

⋆ ⋆ ⋆

A. Einführung und Notation

Die Visualisierung des MVC-Mechanismus ist die spezialisierte Visualisierung eines Entwurfsmusters. Auch die Layoutmuster (Abschnitt 6.1.3) und Verhaltensmuster (Abschnitt 6.3.1) wurden auf eine für sie spezielle Weise visualisiert. Die spezielle Visualisierung erlaubt ein intuitiveres Vorgehen, als es bei einer allgemeinen Entwurfsmuster-Unterstützung wie sie in Abschnitt 6.6 vorgestellt wird, möglich ist.

Jede der beteiligten Klassen `Model`, `View` oder `Controller` wird durch ein eigenes Piktogramm visualisiert. Zwischen ihnen können Verbindungen gezogen werden, die die jeweiligen Abhängigkeitsbeziehungen darstellen (Abbildung 6.20).

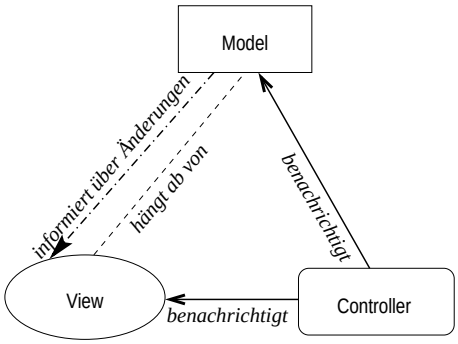


Abbildung 6.20: Notation für den MVC-Mechanismus

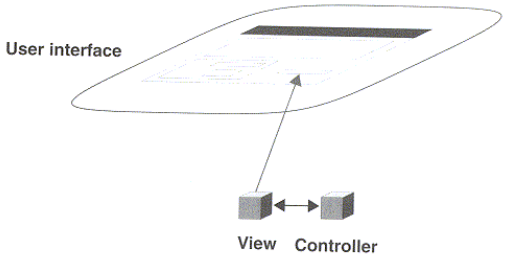
Die Verbindungen müssen gemäß der jeweiligen Realisierung durch eine Klassenbibliothek oder eine Programmiersprache interpretiert werden. Dies geschieht z. B. bei der Realisierung mit der JavaSwing-Klassenbibliothek durch Zusammenfassen von `View` und `Controller` zu einer Klasse (siehe Abschnitt 4.3.3).

⊙⊙⊙

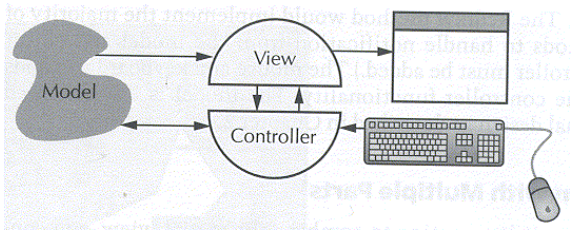
B. Konzeptuelles Modell und Programmierebene

Diese Notation visualisiert einen Teil der struktur- und mechanismenorientierten Ebene (Ebene 2). Im Gegensatz dazu werden die Mechanismen und Strukturen, die in der Verhaltens- und Layoutbibliothek dargestellt werden, aufgabenorientiert visualisiert.

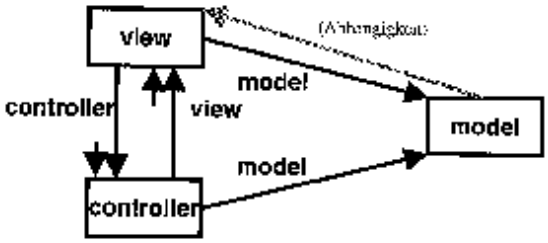
Die Notation folgt bekannten Erklärungen des MVC-Mechanismus (Erklärungsansatz): Wie schon in Abschnitt 2.1.3 ausführlich beschrieben, wird der MVC-Mechanismus oft anhand von Bildern erklärt, z. B.:



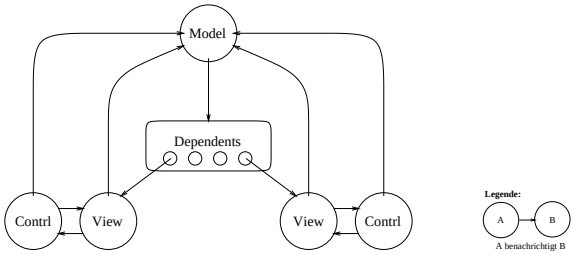
aus dem VisualWorks Tutorial [Par95]



aus Olson [Ols98]



aus Hopkins [Hop97]



aus den Vorlesungsunterlagen des Fachgebiets Programmiersprachen und Übersetzer

Abbildung 6.21: Verschiedene Skizzen als Erklärung des MVC-Mechanismus

⋈⋈⋈

C. Besonderheiten und Bewertung

Die Notation ist implementierungsunabhängig. Eine Entwickler/in, die das Prinzip verstanden hat, kann das für eine Benutzungsschnittstelle zentrale Verhalten des MVC-Mechanismus immer wieder anwenden. Die Notation schließt auch die Lücke zwischen der aufgabenorientierten Visualisierung (Ebene 3) und textueller Programmierung des MVC-Mechanismus (Ebene 1). Dies wurde mit Hilfe einer Programmieraufgabe, die eine Versuchsperson lösen sollte, erfolgreich getestet (siehe Abschnitt 7.4).

∼∼∼

D. Realisierung im Werkzeug COMBO

Der MVC-Editor ist im Rahmen eines studentischen Praktikums implementiert worden [RS00] und realisiert die Notation genau wie unter **A.** beschrieben.

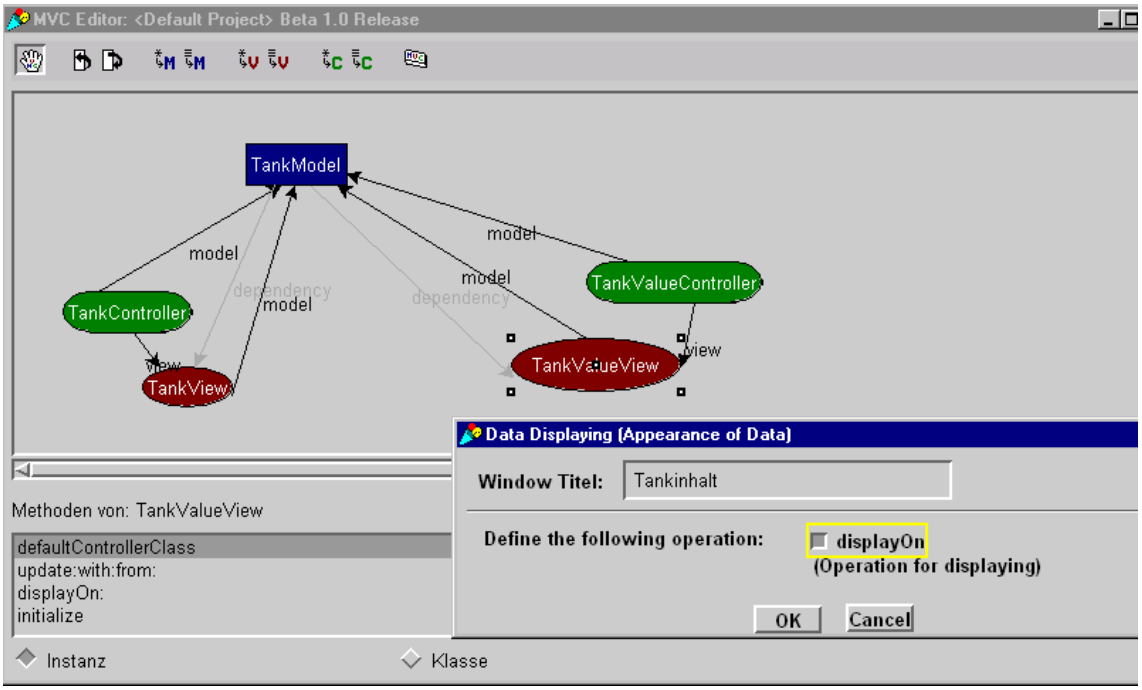


Abbildung 6.22: Der MVC-Editor

Zur Implementierung: Die Verbindungen werden im Rahmen des MVC-Mechanismus von VisualWorks interpretiert. Die Verbindungen zwischen Model und View werden durch den change-update-Mechanismus implementiert, wobei die Entwickler/in durch Assistenten geleitet wird. Mit Hilfe der Assistenten wird z. B. festgelegt, welche Methode den change-update-Mechanismus auslöst.

Diese Realisierung leitet auch zur textuellen Programmierung über, indem bei der Führung der Benutzer/innen an den entsprechenden Stellen Methodenstöberer geöffnet werden. Es wird jeweils erklärt, warum eine Methode in diesem Kontext definiert werden muß (das Beispiel in Abschnitt 7.4 auf Seite 193 zeigt weitere Assistenten und die Klassenstöberer). Es wäre auch möglich, an dieser Stelle zu einer visuelle Notation, z. B. die Objekt-Petrinetze (siehe Abschnitt 6.3.2) zu wechseln. Der Umgang mit COMBO hat aber gezeigt, daß dies ein guter Anknüpfungspunkt für die textuelle Programmierung ist. Somit schließt sich hier die in der Einleitung auf Seite 1.1 beschriebene Lücke.

6.4 Visuelle Methoden für die *Verbindung von Oberfläche und Anwendung*

Wie in Kapitel 5 bereits erwähnt, ist der Aspekt „Verbindung zwischen Benutzungsoberfläche und Anwendung" auch ein Teil des Aspekts „Verhalten", da das Verhalten der Oberflächenelemente durch die Ausgaben der Anwendung beeinflußt werden und die Anwendung mit den Eingaben der Benutzer/in neue Berechnungen ausführt.

Daher können die visuellen Spezifikationsmethoden für den Verhaltensaspekt (z. B. Zustands-Übergangsdiagramme, Sequenzdiagramme oder Objekt-Petrinetze) auch für die Spezifikation der Verbindung verwendet werden.

Intuitiv stellen sich Entwickler/innen diesen Teilaspekt jedoch als eigenen Aspekt vor. Das liegt einerseits daran, daß die durch die Architektur vorgegebenen Komponenten Benutzungsschnittstelle und Anwendung verknüpft werden müssen. Andererseits zeigt sich, daß die Vorgehensweisen zur visuellen Spezifikation der Verbindung in den verschiedenen Methoden alle auf den gleichen Grundprinzipien beruhen; nämlich auf dem Auswählen von Elementen aus mehreren Sichten und dem textuellen (mündlichen) Erklären der Bedeutung der Verbindungen. Diese zwei Prinizipien wurden in Abschnitt 2.3 „Auswählen und Beschreiben" (Seite 39) und „Unterstützung der Verwendung mehrerer Sichten" (Seite 37) genannt.

Die Bedeutung dieser Technik wurde durch die Untersuchung in Abschnitt 3.3.5 bestätigt, daher sollte sie durch Werkzeuge unterstützt werden.

Für das Programmieren der Verbindung von Benutzungsschnittstelle und Anwendung ist „Auswählen und Beschreiben" besonders intuitiv anwendbar, da viele Elemente von Benutzungsschnittstellen Attribute der Anwendung darstellen. Deshalb wurde „Auswählen und Beschreiben" als eigene Spezifikationsmethode in das Werkzeug COMBO integriert.

6.4.1 Auswählen und Beschreiben

⋆ ⋆ ⋆

A. Einführung und Notation

„Auswählen und Beschreiben" kann in vielen Ausprägungen (siehe Abschnitt 2.3) verwendet werden, so daß eine einheitliche Notation nicht sinnvoll erscheint. Es sind aber grundlegende Notations*arten* für bestimmte Interaktionsformen und die Darstellung der Ergebnisse denkbar:

Interaktionsformen für „Auswählen und Beschreiben"

Um kenntlich zu machen, daß eine bestimmte Interpretation für „Auswählen und Beschreiben" erfolgt, muß es möglich sein, die Interpretation einzuschalten. Dies geschieht durch Anwahl eines Modus (z. B. durch Anklicken eines bestimmten Symbols in einer Werkzeugleiste oder durch Auswahl in einem Menü) oder durch automatisches Erkennen bei bestimmen Benutzer/innen-Interaktionen (siehe Abbildungen 6.23 und 6.24). Wenn z. B. ein Element in einem Editor angeklickt ist, kann dieser Editor mit dem Zeiger verlassen werden und ein Element in einem anderen Editor angeklickt werden, was eine Relation zwischen den beiden Elementen herstellt.

Abbildung 6.23: Auswahl eines Modus aus der Werkzeugleiste

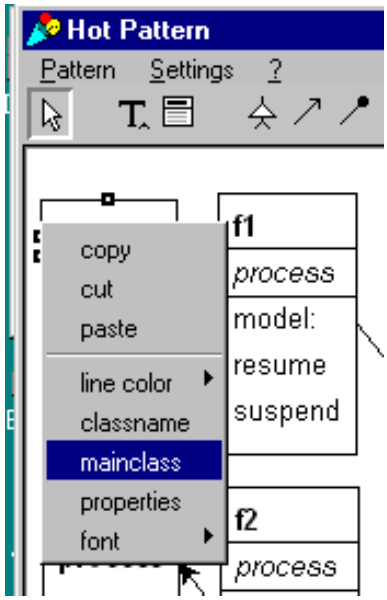


Abbildung 6.24: Auswahl eines Modus durch ein Menü

Darstellung der Ergebnisse von „Auswählen und Beschreiben"

- Elemente, die durch „Auswählen und Beschreiben" miteinander verknüpft wurden, können sichtbar durch Linien oder Pfeile verbunden werden. Dies geschieht z. B. im Werkzeug PARTS [Mai98, Cin00], VisualAge [IBM00] oder auch im Komponentenwerkzeug LCL, das Teil von COMBO ist (siehe Abschnitt 6.7.1).

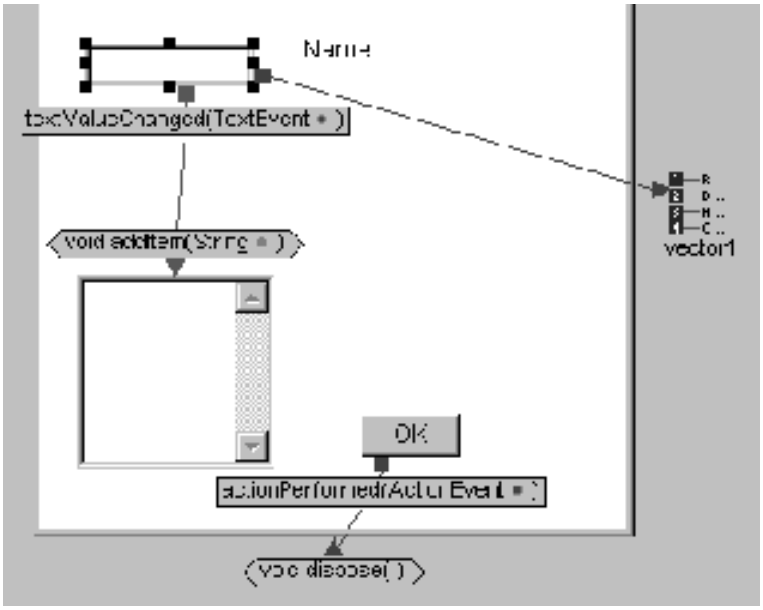


Abbildung 6.25: Pfeile als Darstellung der Beziehungen in PARTS für Java

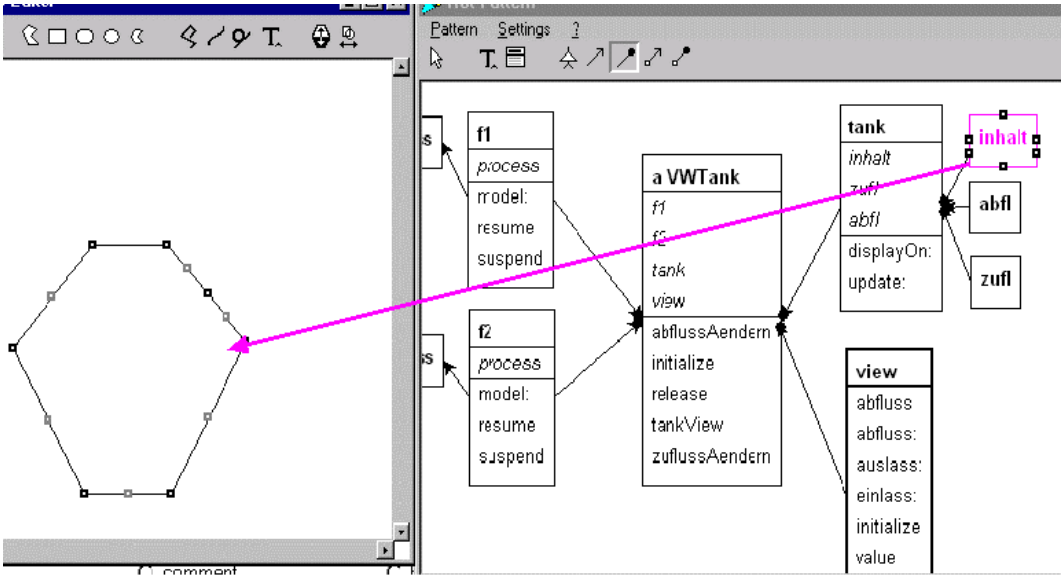


Abbildung 6.26: Pfeile als Darstellung der Beziehungen in COMBO

- Elemente, die durch „Auswählen und Beschreiben" miteinander verknüpft wurden, können mit sichtbaren Notizen versehen werden, wie bei Liebermann in [Lie93a] beschrieben. Gamma notiert die Rollen, die Klassen in Entwurfsmustern spielen, ebenfalls durch Notizen (siehe z. B. [Gam99]).

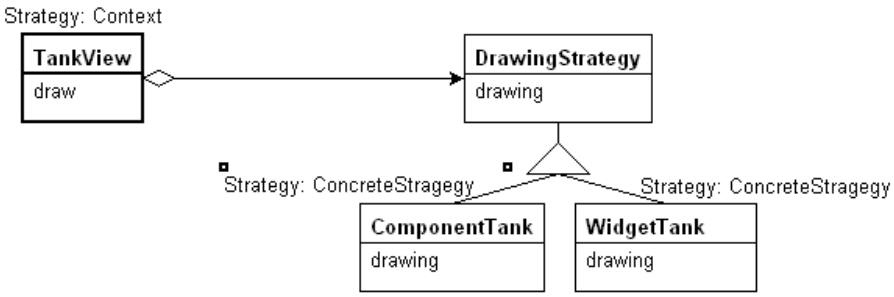


Abbildung 6.27: Beschreibung der Beziehung zwischen Klassen durch Notizen

⊙⊙⊙

B. Konzeptuelles Modell und Programmierebene

Das konzeptuelle Modell für diese Methode beruht auf der Vorstellung, daß die Beziehung zwischen Elementen durch Gesten deutlich gemacht werden.

Je nachdem, wie die Interpretation der Geste ist, erfolgt die Programmierung auf der aufgabenorientierten Ebene (Ebene 3) oder auf der strukturorientierten Ebene (Ebene 2). Dabei können in beiden Fällen auch Elemente aus verschiedenen Ebenen miteinander verbunden werden, z. B. die Elemente der „Baupläne" (d. h. im Klassendiagramm) mit Elementen im Anordnungswerkzeug.

⋈⋈⋈

C. Besonderheiten und Bewertung

Diese visuelle Spezifikationsmethode ist eine Umsetzung der in der Untersuchung in Abschnitt 3.3.5 beobachteten Vorgehensweisen. „Auswählen und Beschreiben" wird als Prinzip in vielen Werkzeugen eingesetzt, allerdings

- wird das Prinzip oft nur sporadisch eingesetzt und nicht konsequent zur Unterstützung der beobachteten Vorgehensweise eingesetzt, und
- es werden bestehende Implementierungsprobleme nicht gelöst (z. B. „Auswählen und Beschreiben" für Elemente in mehreren Fenstern).

Daher ist das Potential, das diese Methode bietet, nicht voll genutzt.

∼∼∼

D. Realisierung im Werkzeug COMBO

Um Auswählen und Beschreiben verwenden zu können, müssen mehrere Bedingungen erfüllt sein:

- Es müssen Elemente (eventuell in mehreren Fenstern) auswählbar sein.
- Es muß ein Modus auswählbar sein, der die Aktionen von „Auswählen und Beschreiben" in den unterschiedlichen Ausprägungen interpretiert, wie in Abschnitt 2.3 vorgestellt.

In COMBO wurde der Schwerpunkt auf die prototypische Implementierung von „Auswählen und Beschreiben" in mehreren Fenstern gelegt. Gerade diese Interaktionsform ist in anderen Werkzeugen nicht möglich, da die Realisierung aus unterschiedlichen Gründen kompliziert ist:

- Weil es in vielen Implementierungen von Benutzungsschnittstellen-Werkzeugen keinen Modus für Interpretation von Gesten zwischen mehreren Fenstern gibt.
- Weil es oft von den unterliegenden graphischen Fenstersystemen oder Editorenentwurfsrahmen keine Unterstützung für die Interaktion in mehreren Fenstern gibt.
- Weil es manchmal keine einheitliche Datenstruktur (Repository) für den aktuellen Entwurf der Benutzungsschnittstelle gibt, die die Darstellung in verschiedenen Fenstern verwaltet. Zur Umsetzung in COMBO siehe Kapitel 7.

Diese Probleme wurden gelöst, indem es eine einheitliche Datenstruktur, das sog. COMBO-Modell gibt, das die Sichten konsistent hält und durch Erweiterungen des HotDraw-Anwendungsrahmens.

In COMBO erfolgt die Auswahl des Modus durch Elemente der Werkzeugleiste. Die Ergebnisse werden nicht explizit in der visuellen Spezifikation festgehalten, sondern es wird Programmtext generiert. Versuche mit Notizen oder Pfeilen wurden sehr unübersichtlich, da Pfeile in vielen der Editoren bereits fest definierte Bedeutungen haben, und die Notizen ohne Verbindungen und Pfeile oft nicht zuzuordnen sind. Daher werden die Relationen, falls nötig, durch Attributfenster definiert. Sie sind dann nur noch sichtbar, wenn das Attributfenster erneut geöffnet wird.

6.5 Visualisierung von Anwendungsrahmen

Alle bisher und im folgenden genannten visuellen Entwicklungsmethoden für objektorientierte Systeme sind auch für Anwendungsrahmen denkbar, z. B. die Darstellung der Objektstruktur, die Darstellung von Entwurfsmustern, die im Anwendungsrahmen verwendet werden, oder die Darstellung von Verhalten einzelner Elemente. Daneben sind auch Visualisierungsformen denkbar, die genau auf einen Anwendungsrahmen zugeschnitten sind.

Beispiel: HotDraw

Der Anwendungsrahmen *HotDraw* [Bra95] dient zur Erstellung zweidimensionaler graphischer Editoren, wie z. B. Klassendiagramm-Editoren. HotDraw ist *das* klassische Beispiel für Anwendungsrahmen, in dem Entwurfsmuster verwendet werden [Joh92, Gam99]. Der Einsatz von HotDraw kann an den folgenden Stellen visuell unterstützt werden:

- Eine Interaktionsmöglichkeit, die sich dann ergeben würde, wäre, wie bei Entwurfsmustern (siehe Abschnitt 6.6), den Entwurfsrahmen als ein Muster anzuzeigen und dann ebenfalls einen Vermischungsmechanismus anzuwenden. Solch eine zusätzliche Visualisierung würde jedoch in der Regel viele Klassen und ihre Verbindungen zusätzlich auf dem (begrenzten) Bildschirmplatz unterbringen müssen und daher zwar zu einer „Spaghetticode"-Darstellung, aber nicht zum Verständnis beitragen.

- Der Entwurfsprozeß selbst könnte durch eine visuelle Darstellung der Abläufe geleitet werden. Im Zusammenhang mit dieser Arbeit hat Jacob eine Sammlung von *Vorgehensmustern* erstellt [Jac00], die die einzelnen Schritte zum Ableiten und Verwenden der Rahmenklassen genau dokumentieren. Solche Muster unterscheiden sich von den in Abschnitt 6.6 behandelten Entwurfsmustern dadurch, daß sie nicht Muster des Systems beschreiben, sondern die Aufgaben der Entwickler/in. Wie in einer Entwurfsmustersprache ist jedoch auf der Anwendung der Vorgehensmuster eine Reihenfolge definiert, die visualisiert werden kann. Dies könnte auch durch eine Navigation mittels einer visuellen Darstellung der einzelnen Schritte unterstützt werden, wie es beispielsweise im Werkzeug autoCAID [Züh00] der Fall ist.

- Zweidimensionale Editoren erstellen oft Graphen, die aus Knoten und Kanten bestehen. Knoten und Kanten haben dabei eine anwendungsspezifische Darstellung. Solche Knoten und Kanten bestehen bei HotDraw aus den Klassen `Figure` und `CompositeFigure`. In der aktuellen Version von HotDraw (Winter 2000) müssen sie textuell programmiert werden. Besser wäre es, sie könnten mit Hilfe einer Zeichenfunktionalität (siehe 6.1.2) festgelegt werden.

- Weiterhin könnte eine Palette von bereits vorhandenen `Figure`- und `CompositeFigure`-Klassen angeboten werden, die eine Benutzer/in geeignet zusammenstellen könnte.

- Die Reaktionen auf Eingaben werden in HotDraw bereits durch eine Zustandsmaschine spezifiziert, die visuell als Zustandsgraph dargestellt ist.

Die angeführten Vorschläge zeigen, daß die Visualisierungen entweder sehr auf den Anwendungsrahmen HotDraw zugeschnitten sind, oder allgemeine Visualisierungen sind, die nicht nur bei Anwendungsrahmen verwendet werden.

Deshalb, und um den Rahmen der Arbeit nicht zu sprengen, werden Visualisierungen von Anwendungsrahmen in dieser Arbeit nicht weiter betrachtet, sondern der Schwerpunkt auf visuelle Spezifikationsmethoden für Entwurfsmuster und Komponenten gelegt.

6.6 Methoden zum Integrieren von Software Engineering-Entwurfsmustern bei der Spezifikation von Benutzungsschnittstellen

Lösungen, die im Rahmen von Entwurfsmustern angegeben werden, sind abstrahierte Vorschläge, die jeweils an das aktuelle Problem angepaßt werden müssen. Dabei kann die aktuelle Implementierung einer Lösung ganz anders aussehen, als die skizzierte Implementierung in einem Entwurfsmusterkatalog, sowohl von der Funktions- oder Klassenstruktur als auch von den Kontrollmechanismen her. Daher ist es schwierig, den Einsatz von Entwurfsmustern durch ein Werkzeug zu unterstützen.

Es ist aber zu beobachten, daß in vielen Fällen von erfahrenen Entwickler/innen eine Implementierungsskizze benutzt wird, um Struktur und Mechanismen als Lösung für das aktuelle Problem zu implementieren (Ebene 1). In der Regel wird dabei in folgenden Schritten vorgegangen:

1. Finden eines Musters, das das aktuelle Problem löst.
2. Identifizieren von *Rollen*, die bereits bestehende Klassen und Objekte in dem gefundenen Entwurfsmuster übernehmen können.
3. Die entsprechenden Methoden und Attribute zu diesen Klassen hinzufügen, damit sie diese Rollen übernehmen.
4. Hinzufügen von neuen Klassen (mit Methoden, Attributen und Beziehungen), die die restlichen Rollen in den Entwurfsmustern übernehmen.

Dieses Vorgehen kann methodisch unterstützt werden:

Schritt 1 kann durch Entwurfsmusterkataloge unterstützt werden, z. B. in Buchform [GHJV95, BMR+96, CNM95] oder elektronisch [GHJV98]. Einige Programmierumgebungen bieten Unterstützung an, indem nach Auswahl eines Entwurfsmusters Programmtextschablonen für Klassen und Methoden des Musters angelegt werden. Die generierten Klassen werden also nicht als Rollen für bestehende Klassen integriert, sondern unparametrisiert als Programmtext niedergeschrieben.

Für die Schritte 2. bis 4. gibt es Ansätze, Entwurfsmuster durch Spracherweiterungen zu unterstützen, z. B. [RDO98, Bos98, Sou97]. Eine Klasse kann, z. B. durch Erben, eine Rolle übernehmen. Dazu ist eine Programmiersprache erforderlich, die Mehrfacherbung erlaubt.

Im Rahmen dieser Arbeit wurden zwei Methoden entworfen, die die Schritte 1. bis 4. visuell unterstützen: *Entwurfsmusterassistenten* und *visuelles Zuordnen von Entwurfsmustern,* die im folgenden beschrieben werden.

6.6.1 Erweiterbarer Entwurfsmusterkatalog und Entwurfsmusterassistenten

⋆⋆⋆

A. Einführung und Notation

Entwurfsmusterkataloge sind am Anfang dieses Abschnitts bereits beschrieben worden.

Entwurfsmusterassistenten [SJ99] sollen die Entwickler/in beim Verwenden von Software-Entwurfsmustern anleiten. Durch Verwenden von Schablonen und visuellen Notationen soll die Entwickler/in durch die

- *Auswahl* des Entwurfsmusters,
- *Entscheidungen* über die beteiligten Klassen, ihre Rollen im Muster und im Rahmen des Musters benötigte Methoden sowie
- die *Parametrisierung* (z. B. Verwendung von Namen für bestimmte, im Muster benötigte Methoden)

geführt werden.

Die Schablonen werden in der Regel fensterbasiert realisiert, d. h. jede Schablone wird durch ein Fenster dargestellt. Es ist jedoch auch eine Flußnotation, z. B. ähnlich den Programmablaufplänen denkbar. Als visuelle Notation wird die aus Abschnitt 6.2 bereits bekannte UML-ähnliche Notation für die Klassen- und Objektstruktur verwendet.

⊙⊙⊙

B. Konzeptuelles Modell und Programmierebene

Das konzeptuelle Modell beruht auf dem in Abschnitt 2.3 vorgestellten Assistentenprinzip: In einer Schritt-für-Schritt-Anleitung wird die Entwickler/in durch die wichtigen Entscheidungen geführt. Ihr werden die möglichen Alternativen vorgestellt. Außerdem wird gezeigt, welche Informationen (Namen von Methoden, Klassen oder ähnliches) an einem bestimmten Punkt im Ablauf benötigt werden, und diese Paramter werden erfragt. Es wird auf der Strukturebene (Ebene 2) programmiert.

⋈⋈⋈

C. Besonderheiten und Bewertung

Der Vorteil dieser Art des Vorgehens ist, daß der Einsatz von Entwurfsmustern den Anfänger/innen erklärt wird, während sie sie programmieren.

Erfahrene Entwickler/innen werden durch den Vorgang geführt und die mentalen Anforderungen werden verringert.

Da sich die Art der Darstellung an bekannten Entwurfsmusterkatalogen, z. B. [GHJV95, GHJV98], orientiert, findet auch kein Bruch zu bekannten Vorgehensweisen aus diesen Katalogen statt.

∼∼∼

D. Realisierung im Werkzeug COMBO

Die meisten Entwurfsmusterkataloge sind nicht erweiterbar. Für den Einsatz der Kataloge wird diese Eigenschaft oft nicht benötigt. Für Entwickler/innen, die neue Muster finden und aufschreiben wollen, ist es jedoch hilfreich, wenn der Katalog erweiterbar ist. Denkbar ist

auch, daß Kollaborationen von Klassen aufgezeichnet und wiederverwendet werden sollen, auch wenn es sich nicht um ein echtes Entwurfsmuster handelt. Deshalb wurde der Entwurfsmusterkatalog in COMBO erweiterbar gestaltet.

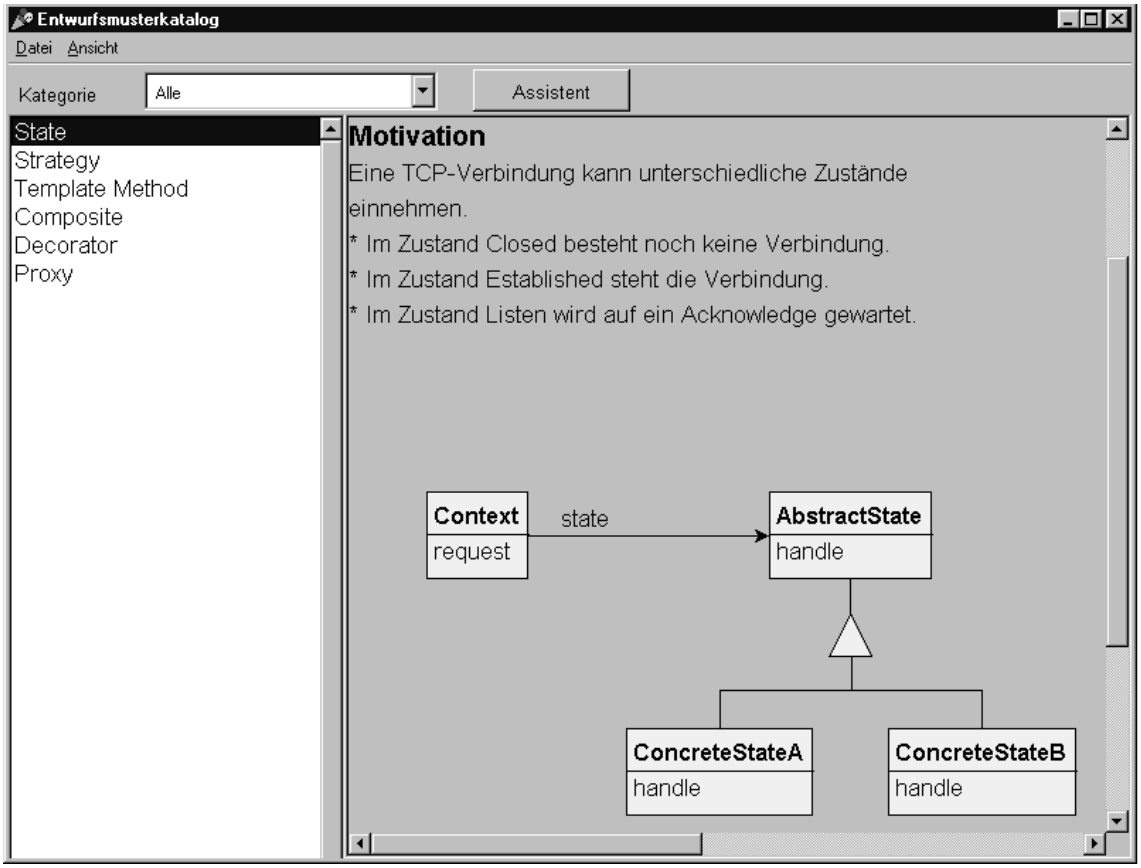


Abbildung 6.28: Der Entwurfsmusterkatalog

Der Entwurfsmusterkatalog und die verschiedenen Entwurfsmuster-Assistenten sind folgendermaßen mit COMBO integriert:

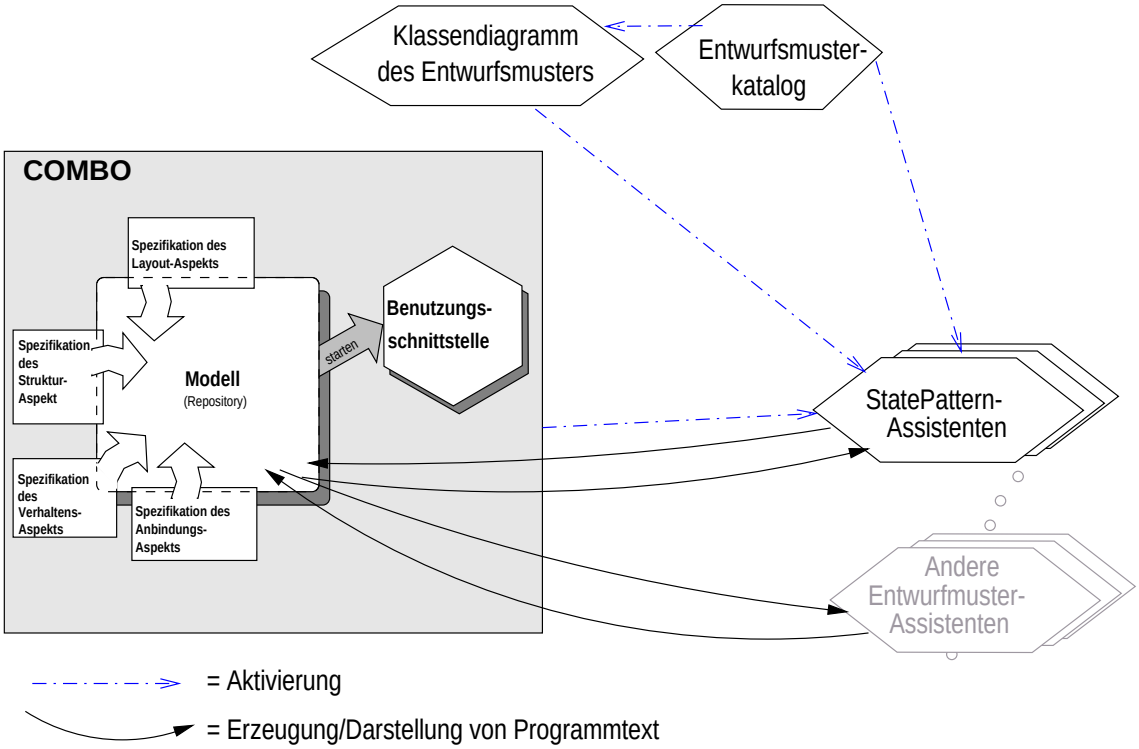


Abbildung 6.29: Zusammenspiel der verschiedenen Assistenten und des Entwurfsmusterkatalogs anhand des Entwurfsmusters „*State*" (*StatePattern*)

Einige Entwurfsmuster-Assistenten und der Entwurfsmusterkatalog wurden in der von mir betreuten Diplomarbeit von Juhnke realisiert [Juh99]:

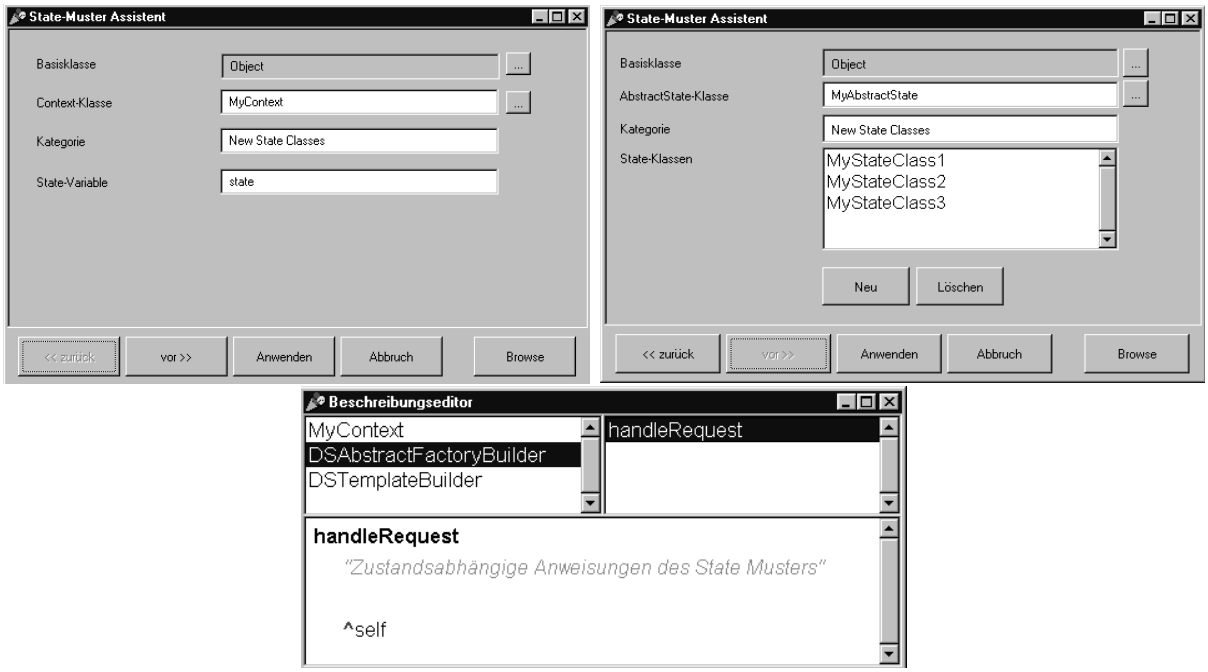


Abbildung 6.30: Die Assistenten für das Entwurfsmuster „*State*"

Die Entwurfsmuster-Assistenten können auch direkt oder aus dem Klassenstöberer von

VisualWorks aufgerufen werden, wie Abbildung 6.31 zeigt.

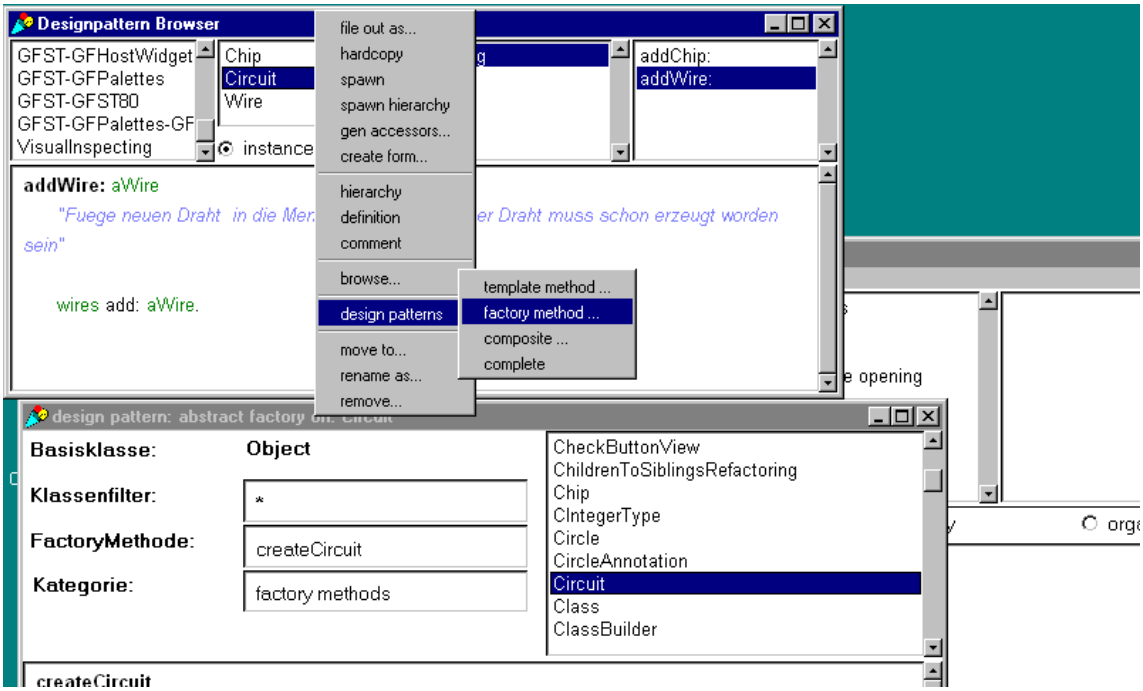


Abbildung 6.31: Die Erweiterung des VisualWorks-Stöberers zum Aufrufen von Entwurfsmuster-Assistenten

6.6.2 Visuelles Zuordnen von Entwurfsmustern

⋆⋆⋆

A. Einführung und Notation

Die Entwurfsmuster-Assistenten führen die Entwickler/in beim Zuordnen der Rollen, die die Klassen im Entwurfsmuster spielen, zu den Klassen, die für die Benutzungsschnittstelle bereits entworfen wurden. Die Vorstellung, daß Klassen bestimmte Rollen im Entwurfsmuster erben, wird jedoch noch deutlicher, wenn sie visuell in einem Klassendiagramm dargestellt wird. Für das Klassendiagramm wird wieder eine UML-artige Notation verwendet, und das Klassendiagramm des Entwurfsmusters wird mit dem Klassendiagramm des aktuellen Entwurfs verschmolzen. Dies kann z. B. durch „Ziehen und Fallenlassen" oder durch „Auswählen und Beschreiben" veranlaßt werden (wie in Abschnitt 2.3 vorgestellt).

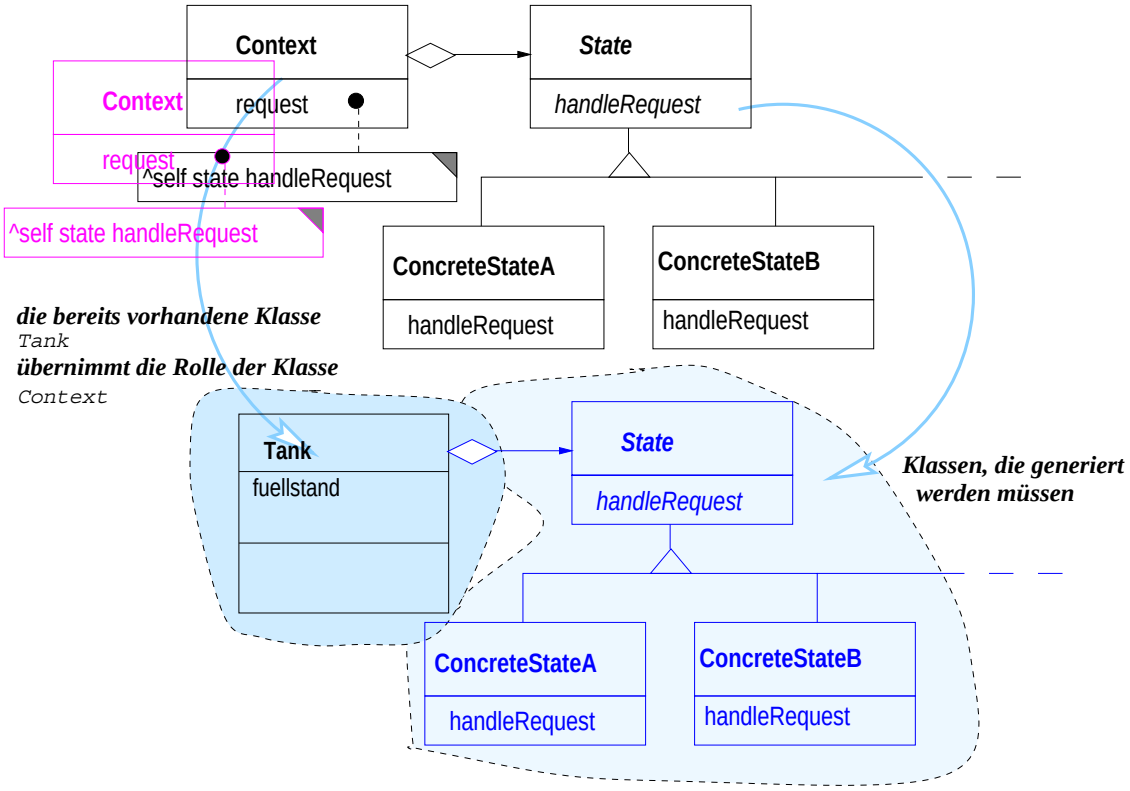


Abbildung 6.32: Zuordnung von Rollen durch „Ziehen und Fallenlassen"

⊙⊙⊙

B. Konzeptuelles Modell und Programmierebene

Die Vorstellung ist, daß Klassen Rollen übernehmen, wie Schauspieler auf der Bühne bestimmte Rollen spielen. Das Annehmen der Rolle ist wie das Anziehen eines Kostüms - durch Überstreifen einer Klasse auf eine andere werden die Eigenschaften der Rolle auf die Basisklasse übertragen. Auf Programmiersprachenebene entspricht das der Verwendung des Sprachkonstrukts „Interfaces" mit automatischer Generierung der Implementierung in Java.

Klassendiagramme sind eine visuelle Darstellung auf der Strukturebene (Ebene 2), daher ist auch das visuelle Zuordnen von Entwurfsmustern auf dieser Ebene angesiedelt.

⋈⋈⋈

C. Besonderheiten und Bewertung

Der Vorteil dieser Methode ist, daß keine neue Darstellung für Entwurfsmuster gelernt werden muß, sondern auf den visuellen Methoden für den Strukturaspekt aufgebaut werden kann. Es hat sich gezeigt, daß die visuelle Spezifikation „visuelles Zuordnen von Entwurfsmustern" dazu führt, daß die Entwickler/innen Muster oder Kollaborationen speichern und wiederverwenden, wie auch beim erweiterbaren Entwurfsmusterkatalog (Abschnitt 6.6.1).

∼∼∼

D. Realisierung im Werkzeug COMBO

Wie besprochen wird die Notation der Klassendiagramme (siehe Abschnitt 6.2.2) verwendet. Zum „Anziehen" der Rollen kann entweder in dem aktuellen Entwurf durch „Ziehen und Fallenlassen" eine Klasse mit einer anderen verschmolzen werden, oder es kann die spezielle

Funktion `merge` verwendet werden, die eine vorher markierte Klasse mit einem ausgewählten Muster verschmilzt. Die zweite Option ist in Abbildung 6.33 und 6.34 dargestellt.

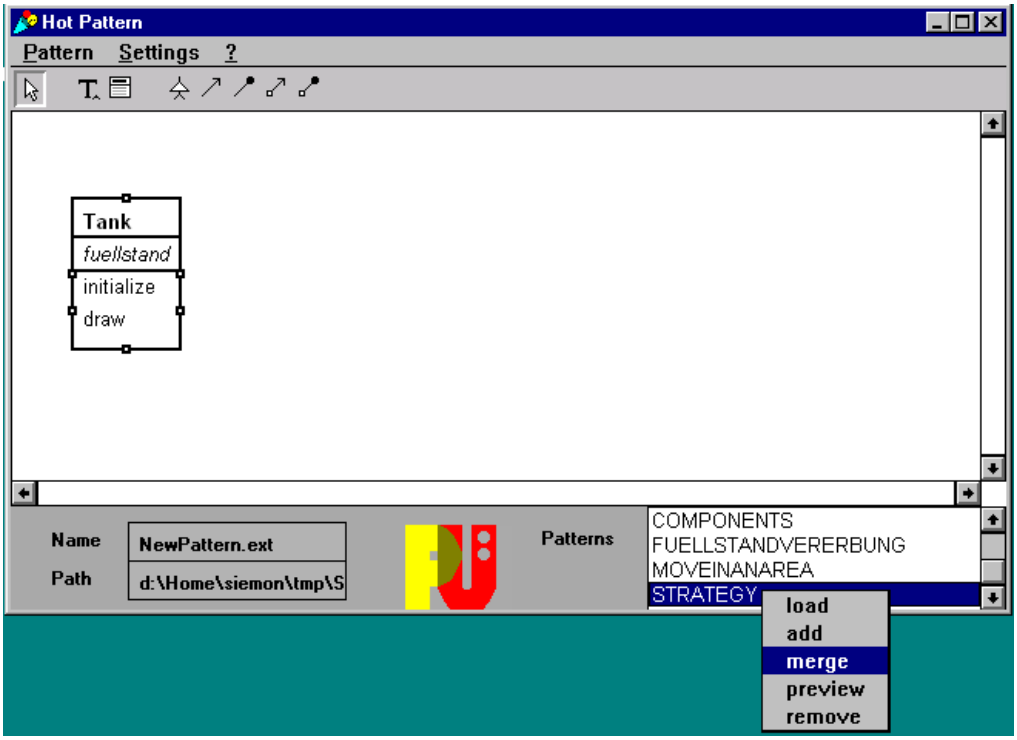


Abbildung 6.33: Anlegen der Klasse `TankView` im Klasseneditor.
Das Entwurfsmuster *„Strategy"* soll mit der Klasse `TankView` verschmolzen werden, indem `TankView` die Rolle der `Kontext`-Klasse übernimmt

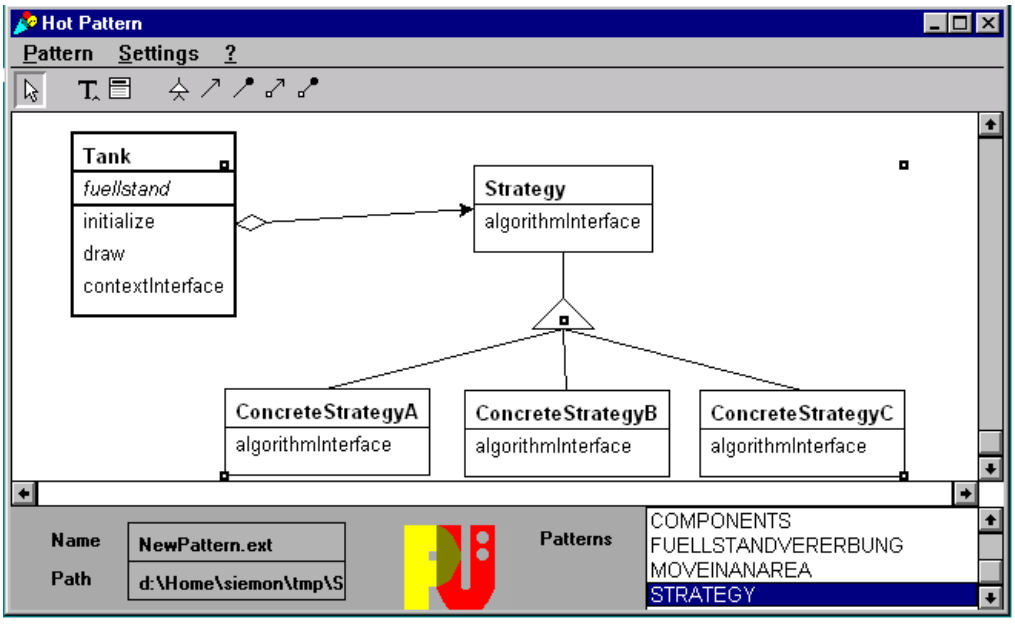


Abbildung 6.34: Die Klasse TankView nach dem Verschmelzen
mit dem Entwurfsmuster *„Strategy"*

6.7 Visuelle Methoden für Komponententechnologie

Wie in Abschnitt 4.4.3 eingeführt, ist das wesentliche Merkmal bei der Verwendung der Komponententechnologie das „Programmieren durch Zusammenstecken". Diese Art von Programmieren ruft bei den meisten Menschen eine Menge an visuellen (und haptischen) Vorstellungen hervor. Solche Vorstellungen gehen vom „Zusammenschrauben aus vorgefertigten Bauteilen" bis zum „Zusammenstecken wie Legobausteine". Das Modell einer Lokomotive, wie sie als Analogie für die Konstruktion in Kapitel 4 vorgestellt wurde, wird in der Regel auch durch Zusammenstecken und Verdrahten konstruiert, wie in Abbildung 6.35 dargestellt.

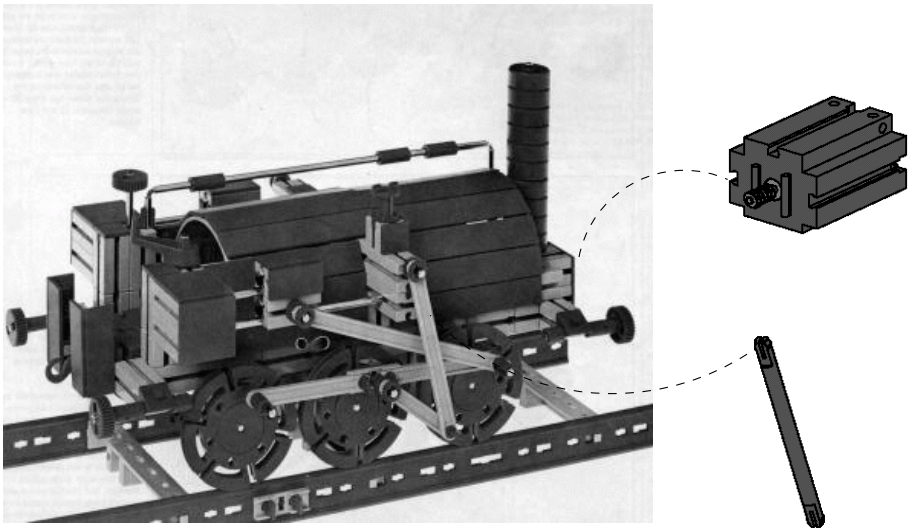


Abbildung 6.35: Das Modell einer Lokomotive kann aus fischertechnik-Bausteinen zusammengesteckt werden (Bilder aus [Kno99, dW99])

Konzeptuelle Modelle für das visuelle Programmieren durch Zusammenstecken von Komponenten sind daher zunächst leicht zu finden. Allerdings passen die Alltagsvorstellungen bei genauerem Hinsehen nicht auf die Programmierung mit Komponenten.

Martin hatte in seiner von mir betreuten Diplomarbeit [Mar00] die Aufgabe, verschiedene Vorstellungen auf ihre Tauglichkeit für die Programmierung von Benutzungsschnittstellen zu untersuchen: Die Lego-, Puzzle und Graphen-Metapher. Als Komponentenmodell wurde das in Abschnitt 4.4.3 vorgestellte Modell verwendet (siehe auch Abbildungen 6.41 und 6.42).

Als Beispiel wurde wieder der Tank verwendet, mit der (in Abbildung 6.36) folgenden Aufteilung in Komponenten [MS00]. Dabei ist zu unterscheiden zwischen (auf der Benutzungsoberfläche) sichtbaren Komponenten, z. B. `tank view` und (auf der Benutzungsoberfläche) nicht sichtbaren Komponenten, z. B. `tank`.

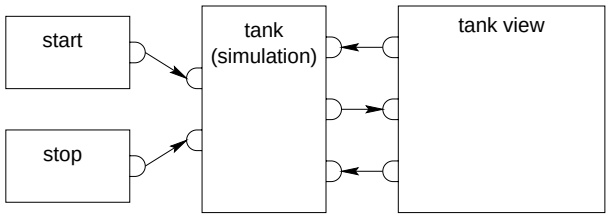
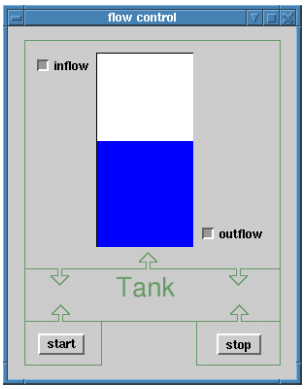


Abbildung 6.36: Mögliche Modularisierung des Tankbeispiels mit Komponenten

Neben der später vorgestellten Graphenmetapher (siehe Abschnitte 6.7.1 und 6.7.2) werden die Vor- und Nachteile der Puzzle- und Lego-Metapher hier kurz präsentiert:

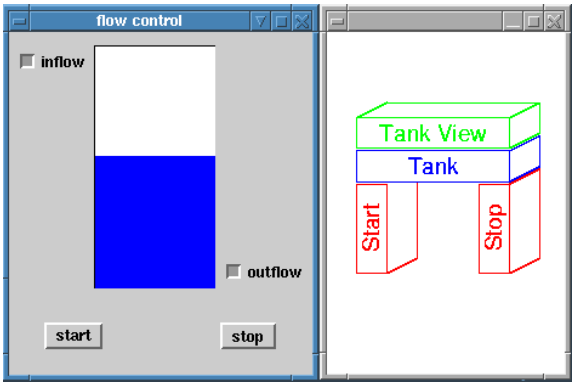
Puzzle-Metapher



Die Puzzle-Metapher ist gut vorstellbar und den meisten Entwickler/innen aus der Kindheit bekannt. Alle Komponenten und ihre Kopplungen sind gleichzeitg zu sehen. Andererseits sind Puzzle-Teile inflexibel, selbst wenn sie in der Größe veränderlich sind und die Puzzle-Nasen und -Buchten verschoben werden können. Es können beispielsweise nicht mehrere Ein- und Ausgänge miteinander verbunden werden, ohne daß sich die Teile überlagern.

Abbildung 6.37: Das Tankbeispiel als Puzzle aus Komponenten
(Bild aus [Mar00])

Lego-Metapher



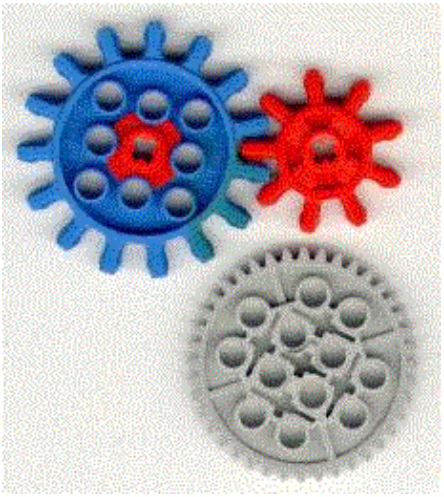
Auch die Lego-Metapher ist wohlbekannt und deshalb ist das Zusammensetzen einer Benutzungsschnittstelle aus „Legosteinen" gut vorstellbar. Um das Layout und die Anordnung der Legosteine darstellen zu können, sind aber zwei Ansichten erforderlich. Legosteine sind durch ihre dreidimensionale Darstellung noch unflexibler als die Puzzle-Teile und brauchen auf dem Bildschirm viel Platz.

Abbildung 6.38: Das Tankbeispiel in der Lego-Metapher
(Bild aus [Mar00])

Die untersuchten Metaphern gingen von gleichartigen Kopplungensarten aus. Eine zusätzliche Schwierigkeit entsteht, wenn die in Tabelle 4.1 auf Seite 99 dargestellten Kopplungsarten

visuell dargestellt werden sollen.

Maschinen-Metapher



Es wäre gut wenn Komponenten wie Bauteile von Maschinen dargestellt werden könnten: Bei Maschinenteilen bestimmt die Funktion die Form. In der nebenstehenden Abbildung wird deutlich, daß das untere Zahnrad nicht ohne Adapter mit den beiden anderen Zahnrädern zusammenarbeiten kann. Es ist aber auch sofort zu sehen, wie eine Achse beschaffen sein muß, auf die eines der Zahnräder gesteckt werden kann.
Bei Softwarekomponenten ist es nicht ganz so einfach: In der Regel ist die Kopplung unabhängig von der Funktionalität. Die Funktionalität einer Softwarekomponente kann auch nur abstrahiert durch eine Form dargestellt werden. Oft ist es aber leichter, die Funktionalität durch die Benennung der Softwarekomponente darzustellen, als durch ein Piktogramm.
Daher ist die Maschinenmetapher zwar geeignet, um in der Person, die ein Bild betrachtet, die Vorstellung hervorzurufen, wie Komponenten miteinander arbeiten. Sie ist aber nicht geeignet, um die Funktionalität und Kopplung von Softwarekomponenten zu beschreiben.

Abbildung 6.39: Kopplung von Maschinenteilen als Metapher für die Kopplung von Komponenten

Auch sollen nicht alle Komponenten mit allen anderen Komponenten gekoppelt werden können. D'Souza [DW98] stellt funktional angepaßte Schnittstellen dar, wie in Abbildung 6.40 zu sehen. Dabei haben die Kopplungen verschiedene Formen. Im Gegensatz zur Maschinen-Metapher dienen die Formen jedoch nur zur Unterscheidung und visualisieren nicht die Funktion der Teile.

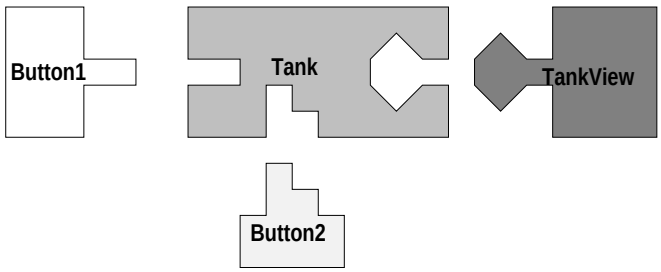


Abbildung 6.40: Funktional angepaßte Schnittstellen

6.7.1 Visuelles Zusammenstecken von Komponenten in der Schichtendarstellung

⋆ ⋆ ⋆

A. Einführung und Notation

Beim visuellen Zusammenstecken werden fertige Komponenten miteinander verbunden. Sichtbare und unsichtbare Komponenten können dabei entweder in einer Darstellung (wie in VisualAge für Java [IBM00], diese Notation benutzt die Methode „Auswählen und Beschreiben", siehe Abschnitt 6.4) oder in mehreren Darstellungen (wie in JavaStudio, siehe Abschnitt 4.5) zusammengesteckt werden. Wird nur eine Darstellung verwendet, liegen die Komponenten manchmal in verschiedenen Schichten (engl. *layers*). Die Kopplung erfolgt durch Pfeile, die, je nach Interpretation, Datenflüsse, Nachrichtenaufrufe oder Ereignisflüsse darstellen.

⊙ ⊙ ⊙

B. Konzeptuelles Modell und Programmierebene

Die Vorstellung ist, daß der Aspekt Anordnung zusammen mit dem Aspekt Verhalten in einer Darstellung spezifiziert werden kann. Dabei wird das Verhalten nicht über Kontrollstrukturen von Programmiersprachen, d. h. Basismechanismen (Ebene 0) definiert, sondern durch Aktivierung des Verhaltens der Komponenten. Eine Komponente hat Ein- und Ausgänge und wenn diese aktiviert werden, wird „in" der Komponente das entsprechende Verhalten angestoßen. Auf einer abstrakteren Ebene ist die Vorstellung die gleiche wie bei den Objekt-Petrinetzen (Abschnitt 6.3.2), wobei eine Komponte mehrere Objekte enthalten kann.

Die Programmierung durch Zusammenstecken erfolgt auf der Strukturebene (Ebene 2), mit Ausnahme der Anordnung der sichtbaren Komponente, die aufgabenorientiert ist (Ebene 3). Um eine aufgabenorientierte Programmierung der Kopplung zu erreichen, müßte eine Darstellung entsprechend der Maschinen-Metapher (siehe oben) gefunden werden.

⋈⋈⋈

C. Besonderheiten und Bewertung

In [Mar00] sind die Vor- und Nachteile der einzelnen Darstellungen ausführlich diskutiert worden.

Zusätzlich werden im Kontext des konzeptuellen Modells auch die Ergebnisse von Abschnitt 4.5.2 verständlich: In Programmierumgebungen wie PARTS oder VisualAge scheint die Benutzer/in auf der Komponentenebene zu programmieren. Die Pfeile bedeuten aber teilweise Methodenaufrufe (Ebene 0), teilweise Komponentenaktivierung (Ebene 2) und teilweise Benutzer/innen-Eingaben (Ebene 3). Das ist verwirrend, zumal die Aufteilung in die Ebenen eine Unterscheidung aus der vorliegenden Arbeit ist, und den Entwickler/innen, die diese Arbeit nicht gelesen haben, u. U. gar nicht so klar ist.

∼∼∼

D. Realisierung im Werkzeug COMBO

Die Ergebnisse aus Abschnitt 4.5.2 und des Vergleichs in [Mar00] haben zu der im folgenden beschriebenen Realisierung geführt, die im Rahmen der Diplomarbeit von Martin implementiert wurde.

Der Realisierung von Martin lag das bereits erwähnte Komponentenmodell zugrunde, dessen Klassenstruktur in Abbildung 6.41 und dessen Kommunikation in Abbildung 6.42 dargestellt ist.

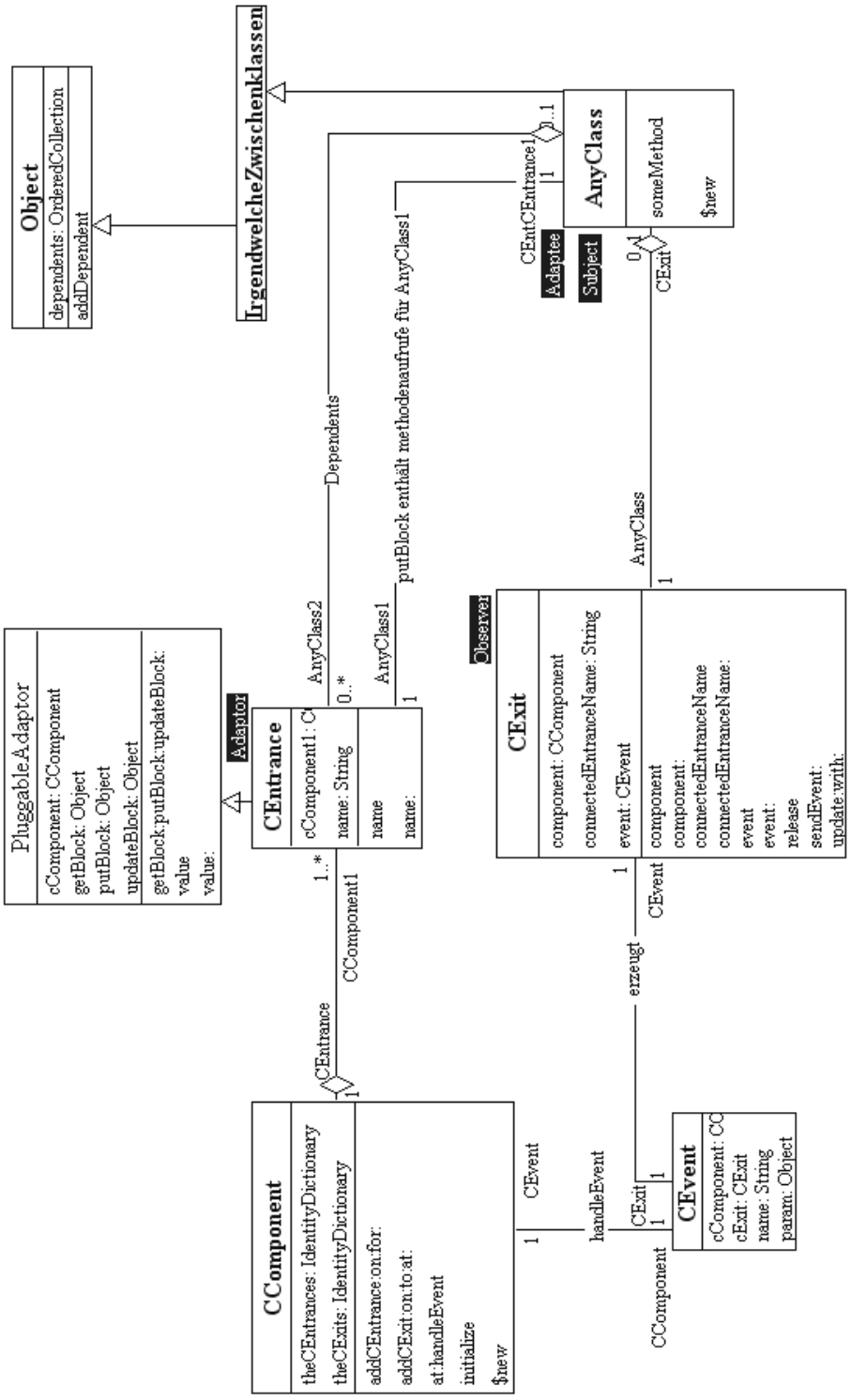
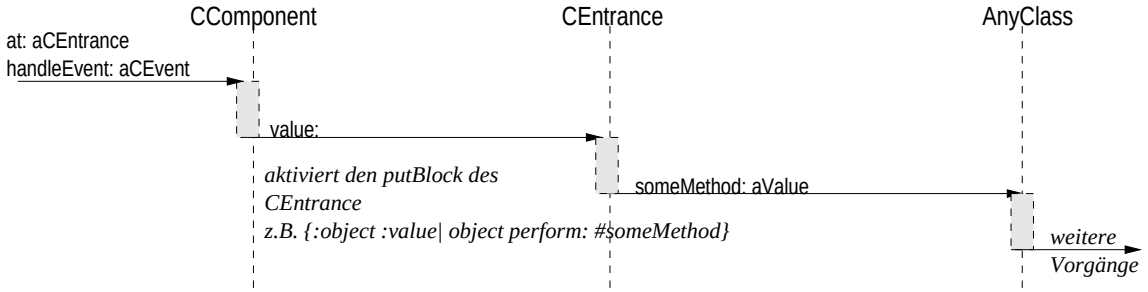


Abbildung 6.41: Das der Realisierung in COMBO zugrundeliegende Komponentenmodell

Verarbeitung eines Ereignisses, das an einem Eingang anliegt



Ein Objekt der Komponente aktiviert einen Ausgang

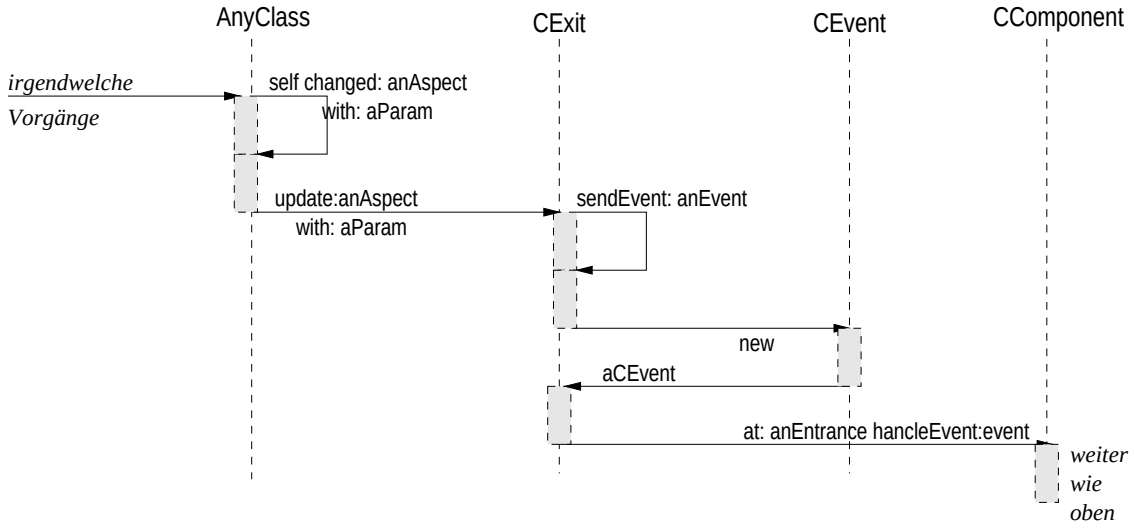


Abbildung 6.42: Kommunikation der Komponenten

Mit diesem Komponentenmodell, das von Martin erweitert wurde, wurde eine Schichten-Darstellungsform gewählt. Darin können sichtbare und unsichtbare Komponenten, sowie die Beschriftungen und Kopplungen in einem Fenster, aber in verschiedenen Schichten dargestellt werden. Dies ist in Abbildung 6.43 zu sehen.

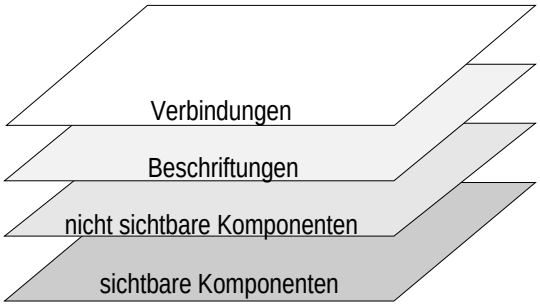


Abbildung 6.43: Die Schichten der Komponenten-Darstellung
(Bild nach [Mar00])

In dieser Darstellung kann der Tank aus der in Abbildung 6.36 gezeigten Komponentenaufteilung wie in Abbildung 6.44 zusammengesteckt werden. Im nebenstehenden Dialog können

die Ein- und Ausgänge explizit ausgewählt werden.

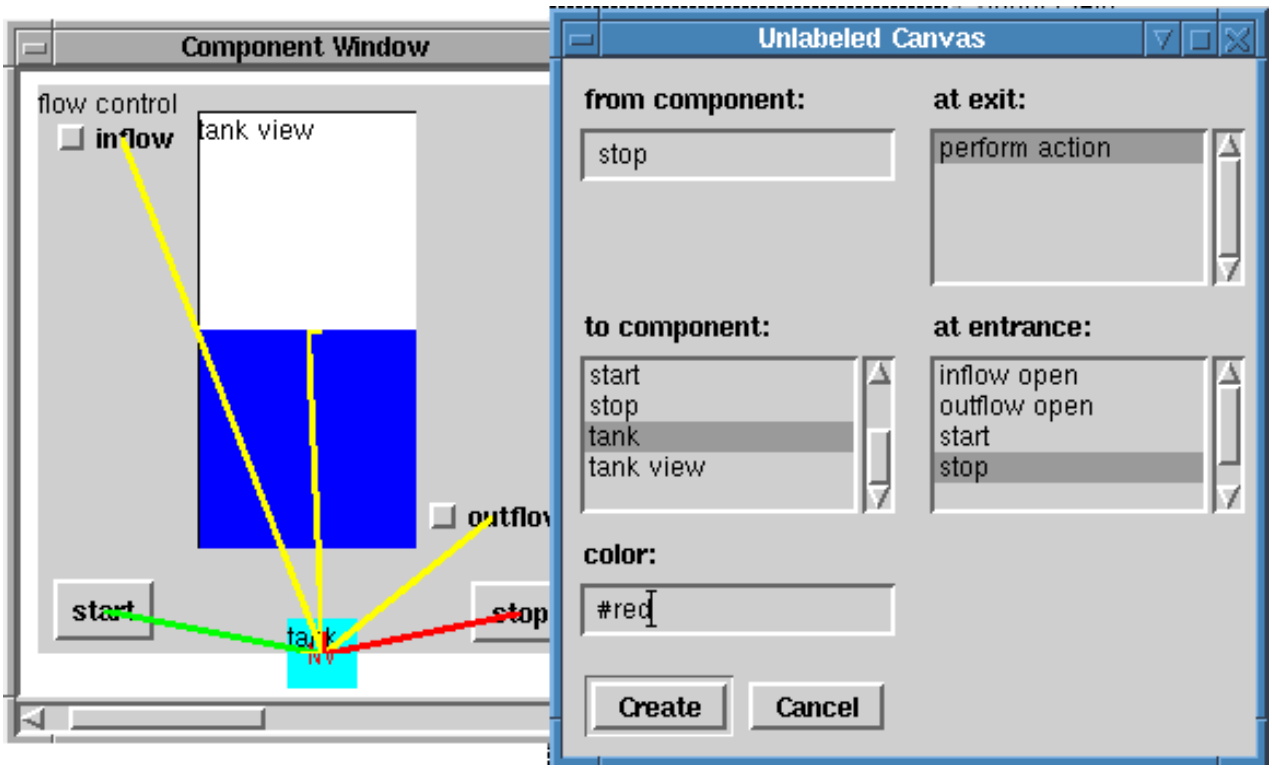


Abbildung 6.44: Die LCL-Darstellung
(Bild aus [Mar00])

6.7.2 Visuelles Zusammensetzen von Komponenten in Chip-Darstellung

⋆⋆⋆

A. Einführung und Notation

Die Chip-Darstellung ist etwas einfacher als die Schichten-Darstellung. Hier wird der Anordnungsaspekt aus der Darstellung ausgeschlossen und nur das Zusammenstecken selbst dargestellt. Die Notation folgt der Metapher des Verdrahtens von (Computer-)Chips: Komponenten sind Kästchen, die Ein- und Ausgänge haben, die miteinander verdrahtet werden können.

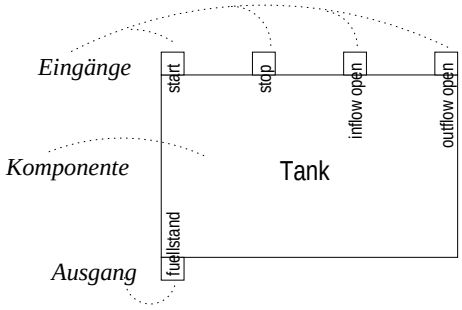


Abbildung 6.45: Notation für die Chipdarstellung

⊙⊙⊙

B. Konzeptuelles Modell und Programmierebene

Computer-Chips sind Komponenten sehr ähnlich, indem sie (abstrakt gesehen) Funktionalität kapseln und gleichartige Ein- und Ausgänge haben, durch die Signale (z. B. zeitlich sich ändernder elektrischer Strom) fließten, deren Analogie ein Strom von Ereignissen ist.

Diese Vorstellung liegt wieder auf der Strukturebene (Ebene 2).

⋈⋈⋈

C. Besonderheiten und Bewertung

Die einfachere Vorstellung, die wieder nur einen Aspekt spezifiziert (Verhalten des Systems als Kopplung aus Komponenten), paßt eher zum Aspektmodell als die Schichtendarstellung. Die Schichtendarstellung hat aber den Vorteil, daß die sichtbaren Komponenten durch ihre Darstellung als Widgets besser „anzufassen" sind. Die Erfahrung hat gezeigt, daß die Entwickler/innen in Abhängigkeit von bereits bekannten Metaphern die Schichten- (wenn sie schon beispielsweise mit VisualAge gearbeitet haben) oder die Chip-Darstellung bevorzugen (wenn sie noch nicht mit einem Komponentenwerkzeug gearbeitet haben).

∼∼∼

D. Realisierung im Werkzeug COMBO

Es wurde eine direkte Umsetzung der unter **A.** beschriebenen Notation realisiert.

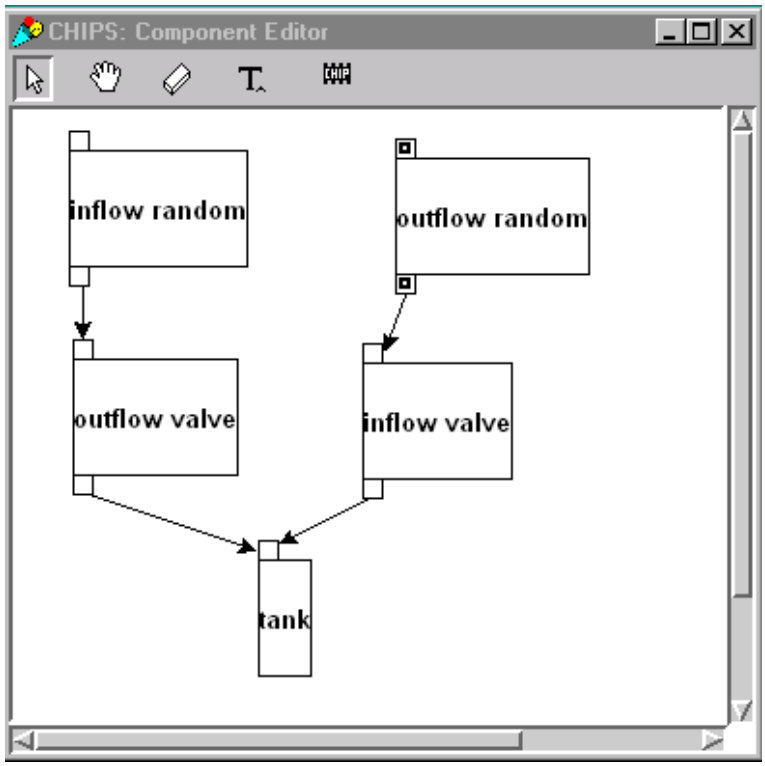


Abbildung 6.46: Verdrahtung von Komponenten für das Tank-Beispiel

Kapitel 7

COMBO: Ein Werkzeug zum visuellen Entwurf von Benutzungsschnittstellen

In Kapitel 6 wurden visuelle Spezifikationsmethoden für die Entwicklung von Benutzungsschnittstellen vorgestellt, die auf dem Erklärungsansatz und dem Aspektansatz aufbauen.

Diese Spezifikationsmethoden sollen gemeinsam verwendet werden können, d. h. daß sie in einen Entwicklungsprozeß integriert sind, der der Entwickler/in bekannt sein muß, und der durch geeignete Navigationshilfen unterstützt wird.

Deshalb wird in diesem Kapitel die **Benutzungsoberfläche von COMBO** vorgestellt, d. h. die **Projektverwaltung** und die **Navigation** zur Unterstützung des Entwicklungsprozesses.

Außerdem wird die Implementierung von COMBO vorgestellt, um Leser/innen, die COMBO erweitern möchten, die Anknüpfungspunkte dafür zur verdeutlichen.

Danach wird untersucht, inwieweit die in Kapitel 2 und 5 geforderte **Durchgängigkeit durch die verschiedenen Programmierebene**n im Werkzeug COMBO gegeben ist.

Zuletzt soll angedeutet werden, welche **Ideen für Erweiterungen** des Werkzeugs COMBO entwickelt wurden, ebenfalls als Anknüpfungspunkte für interessierte Leser/innen und nachfolgende Arbeiten.

7.1 Übersicht

COMBO ist in Smalltalk mithilfe der Entwicklungsumgebung VisualWorks realisiert; alle Benutzungsschnittstellen-Werkzeuge der Standard-Klassenbibliothek von VisualWorks sind mit COMBO integriert (siehe Abschnitt 7.3.1).

Zusätzlich wurde der Entwurfsrahmen HotDraw (siehe [Joh92], [Bra95], sowie [Jac00]) benutzt, um einige graphischen Editoren zu erstellen.

Wie in Kapitel 6 beschrieben, werden die verschiedenen Spezifikationsmethoden als einzelne Editoren realisiert, die durch eine gemeinsame Datenstruktur, das COMBO-Modell, realisiert werden.

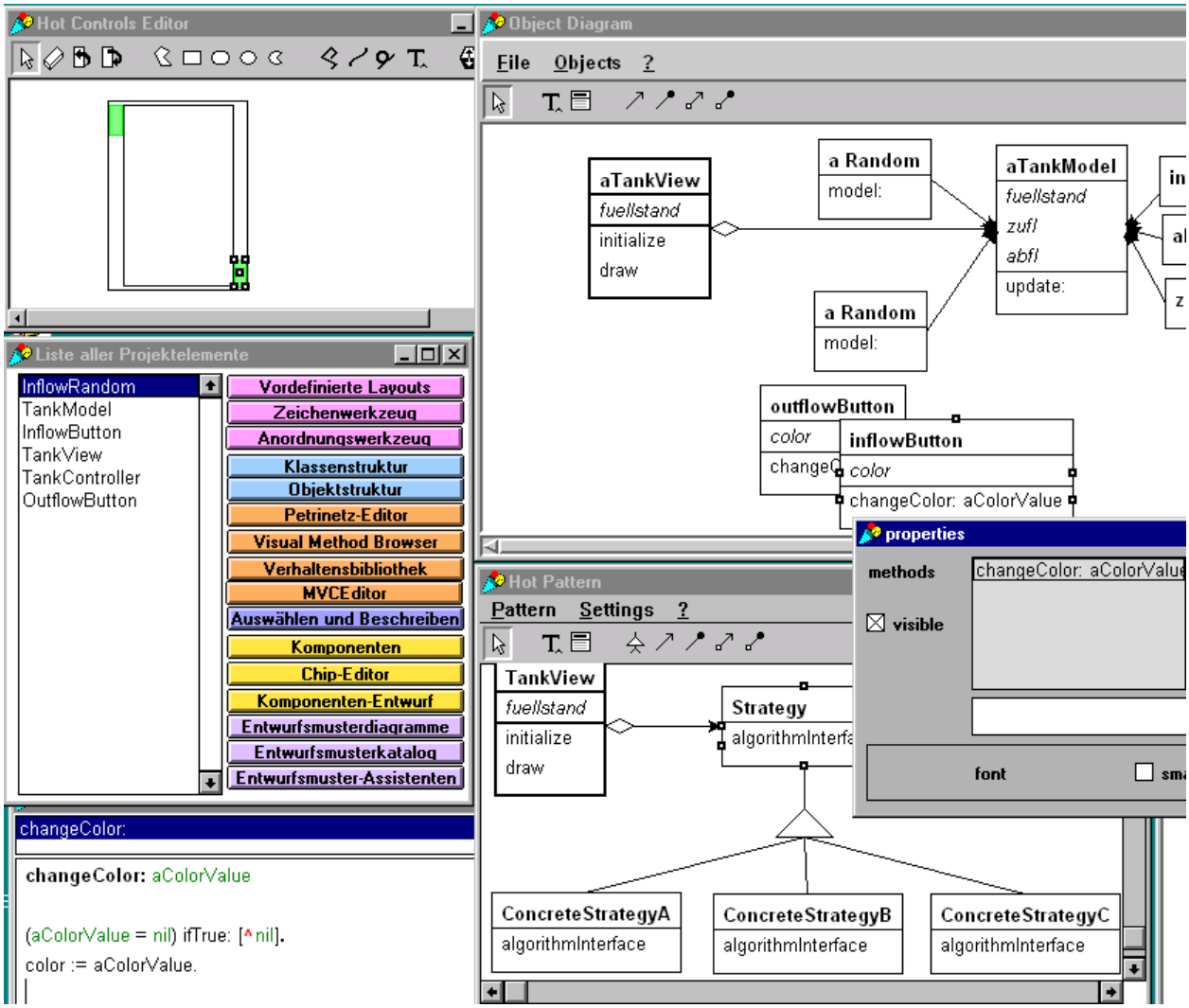


Abbildung 7.1: Übersicht über die Verwendung mehrerer Editoren in COMBO

Der Name des Werkzeugs ist COMBO, in Analogie zu einer kleinen Musikkapelle mit mehreren Instrumenten.
Um eine Aufnahme (Benutzungsschnittstelle) zu erstellen, müssen mehrere Musiker/innen miteinander spielen (Editoren miteinander arbeiten). Sie setzen die Noten mit der Interpretation der Dirigent/in (konzeptuelles Modell der Entwickler/in) um, wobei sie jeweils nur eine Stimme spielen (einen Aspekt spezifizieren).
Die Analogie trifft auf die aspekt-übergreifenden Editoren natürlich nicht mehr zu. Es soll aber deutlich werden, daß das konzeptuelle Modell im Mittelpunkt steht und die Editoren verschiedene Aspekte des Gesamtwerks ausdrücken.

7.2 Benutzungsoberfläche von COMBO

7.2.1 COMBO-Projektverwaltung

Aus der Sicht der Entwickler/in ist die COMBO-Projektverwaltung der Startpunkt für das Arbeiten mit COMBO. Beim Starten von COMBO (z. B. durch Auswahl aus dem Tools-Menü von VisualWorks oder durch Eingabe von `COMBO open`) erscheint eine Liste aller bereits vorhandenen Projekte. Die Entwickler/in kann ein bereits bestehendes Projekt laden oder ein neues Projekt anfangen, siehe Abbildung 7.2.

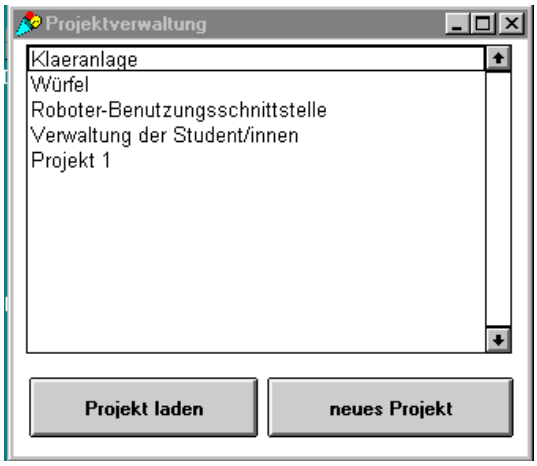


Abbildung 7.2: Die Projektverwaltung zeigt alle COMBO-Projekte an

7.2.2 COMBO-Navigation

Im vorhergehenden Kapitel wurden 16 visuelle Spezifikationsmethoden vorgestellt und Hinweise auf weitere mögliche Methoden gegeben. Außerdem steht im Hintergrund immer auch die textuelle Programmierung, mit weiteren Hilfen wie Stöberern (z. B. in Smalltalk der sogenannte *class browser*). Selbst wenn jede einzelne Methode klar und überschaubar ist, kann die Vielfalt von Methoden Verwirrung erzeugen.

Da bei der Entwicklung die Interaktion in mehreren Fenstern stattfindet, also die Arbeit an der Benutzungsschnittstelle in mehreren Unterwerkzeugen geschieht, ist die Navigation zwischen den einzelnen Werkzeugen sehr wichtig.

Nachdem ein Projekt ausgewählt wurde, erscheint das Navigationsfenster, das Abbildung 6.1 über die Architektur vom COMBO nachempfunden ist, da dies dem konzeptuellen Modell der Bearbeitung entspricht.

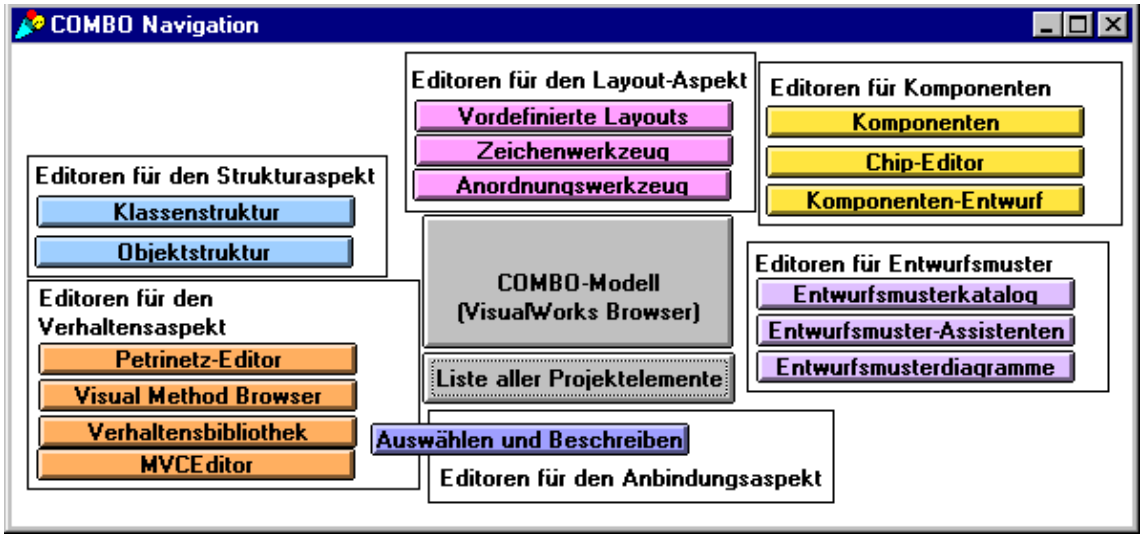


Abbildung 7.3: Navigation zwischen den verschiedenen Entwurfswerkzeugen.

Die Anordnung wurde um das Modell herum gruppiert, um der Entwickler/in zu zeigen, daß es keine Entwurfsreihenfolge zwischen den verschiedenen graphischen Methoden gibt, die durch die Werkzeuge realisiert werden.

Durch Anklicken der einzelnen Werkzeugknöpfe werden die entsprechenden Werkzeuge gestartet, bzw. es wird zwischen den Werkzeugen umgeschaltet.

Bei der Navigation werden zwei Arten von Werkzeugen unterschieden: Solche, die alle Entwurfselemente des Projekts darstellen und solche, die einzelne Klassen bearbeiten.

Beim Anklicken von Werkzeugknöpfen der ersten Kategorie (Klasseneditor, Klassenbrowser) wird das entsprechende Werkzeug mit einer Darstellung aller Elemente des aktuellen Projekts geöffnet, bzw., wenn es bereits geöffnet ist, in den Vordergrund gebracht. Werkzeuge, die einzelne Elemente bearbeiten, werden beim Anklicken mit leerer Arbeitsfläche geöffnet.
Um Werkzeuge für bestimmte Klassen zu öffnen, muß die Liste aller Projektelemente geöffnet, die entsprechende Klasse ausgewählt und das entsprechene Werkzeug über einen Knopf aktiviert werden.

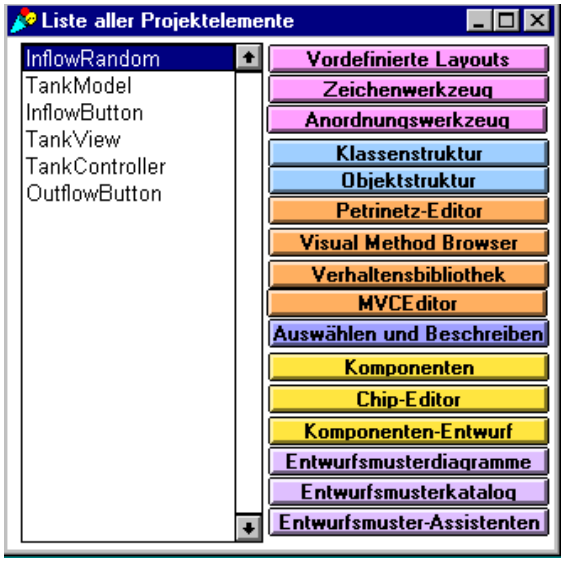


Abbildung 7.4: Die Liste aller Elemente, die zu einem Projekt gehören.

7.3 Implementierung von COMBO

Das System COMBO ist eine Entwicklungs- und Programmierumgebung für Benutzungsschnittstellen. Die Implementierung basiert auf einer Erweiterung der Smalltalk-Programmierumgebung VisualWorks.

Das System COMBO besteht aus verschiedenen Teilen, alle Teile bauen auf der VisualWorks-Klassenbibliothek und dem dazugehörigen VisualWorks-System auf (siehe Abbildung 7.5):

- Das **COMBO Modell** ist das Datenmodell, in dem die zu entwickelnde Benutzungsschnittstelle (der aktuelle Entwurf) gehalten wird. Es besteht aus abstrakten und konkreten Klassen. Die **COMBO Projektverwaltung** ist die Schnittstelle zwischen COMBO Modell und COMBO Editoren. Für die Editoren übernimmt sie die Verwaltung der Einträge ins Modell und für die Benutzungsschnittstellen-Elemente verwaltet sie die Zugehörigkeit zu den COMBO Editoren.

- Die **COMBO Editoren** sind die eigentlichen Spezifikationswerkzeuge

- Die **COMBO Navigation** leitet die Benutzer/in von COMBO, d. h. die Entwickler/in der Benutzungsschnittstelle, durch die verschiedenen Aspekte und Programmierebenen.

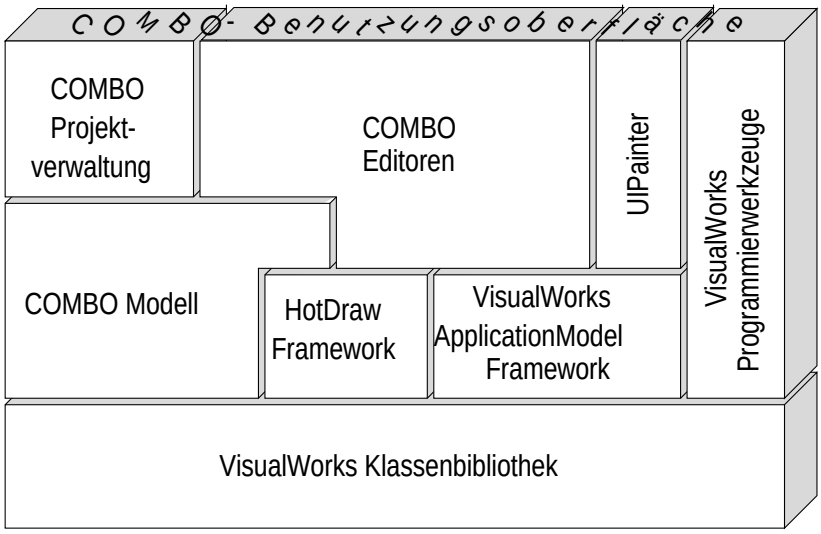


Abbildung 7.5: Die Software-Teile von COMBO

Dieser Abschnitt beschreibt zunächst die Implementierung des COMBO-Modells, mit dessen Hilfe alle Editoren integriert werden. Anschließend wird auf die Editoren eingegangen, wobei insbesondere die Grundstruktur der Editoren, die mit dem HotDraw-Entwurfsrahmen erstellt wurden, kurz vorgestellt werden.

7.3.1 Struktur der Benutzungsschnittstellen: das COMBO-Modell

Für ein Datenmodell, das eine Benutzungsschnittstelle beschreibt, gibt es mehrere Alternativen:

1. Ein neuer Entwurfsrahmen für Benutzungsschnittstellen mit abstrakten und konkreten Klassen, eventuell unter Verwendung einer neuen Modellbeschreibungssprache (dieser Ansatz wurde z. B. für die Entwurfsrahmen ET++ [Gam91] oder HotDoc [Buc98] gewählt). Der Vorteil dieser Alternative ist, daß in den Klassen die Methoden zur Verwaltung eines aktuellen Entwurfs implementiert sein können. Nachteilig ist, daß dann die gesamte Benutzungsschnittstellen-Funktionalität implementiert werden müßte. Auch könnten bereits bestehende Benutzungsschnittstellen-Werkzeuge nicht eingebunden werden, d. h. außer den mit dem Entwurfsrahmen erstellten Bausteinen könnten in einer Benutzungsschnittstelle keine weiteren Bausteine verwendet werden.

2. Das Datenmodell dient nur zum Halten der Klassen und Objekte, die beim Entwurf der Benutzungsschnittstelle verwendet werden. Damit können bereits bestehende Bausteine in den Entwurf eingebunden werden. Außerdem können verschiedenartige Editoren miteinander integriert werden. Die zusätzliche Funktionalität für die Verwaltung kann mithilfe zweier Mechanismen an die Elemente des aktuellen Entwurfs gebunden werden:

   (a) Durch Erbung: Das Datenmodell stellt Klassen (oder abstrakte Klassen wie z. B. in Java die sog. Interfaces) zur Verfügung, die die Verwaltungsfunktionalität implementieren (oder in abstrakten Klassen unimplementiert beschreiben). Alle Elemente des aktuellen Entwurfs müssen dann von diesen Klassen erben. Nachteilig

ist, daß in Sprachen ohne Mehrfacherbung, wie Smalltalk, nahezu alle Klassen für die Benutzungsschnittstellen-Entwicklung erweitert werden müssen.

(b) Mithilfe des Entwurfsmusters „*Dekorator*". Jede Klasse, die in dem Modell, d. h. dem aktuellen Entwurf der Benutzungsschnittstelle, enthalten ist, wird mit einem Dekorator verknüpft, der die Verwaltungs- und alle weiteren benötigten Funktionalitäten implementiert.

COMBO wurde, nach einigen Versuchen mit den anderen Alternativen, wegen der Flexibilität mit Hilfe der zweiten Alternative, unter Verwendung des Dekorator-Musters, realisiert.

Eine Benutzungsschnittstelle, die mit Hilfe von COMBO entwickelt wird, wird Projekt genannt. Jedes Projekt ist eine Sammlung von Benutzungsschnittstellen-Elementen, die durch Dekoratoren gekapselt werden.

Es wurde eine Klasse `COMBOProject` definiert, die die Verwaltung eines Projekts übernimmt.

Da in der VisualWorks Klassenbibliothek das Entwurfsmuster Dekorator in der allgemeinen Form noch nicht enthalten ist, wurde eine Klasse `Decorator` entworfen, die das Muster realisiert.

Die Weitergabe von Methoden, die der Dekorator nicht ausführen kann, sondern an die ursprünglichen Klassen delegiert werden, erfolgt mit Hilfe des Entwurfsmusters „*Proxy*". Fehlermeldungen der Form „`selector not found`" werden abgefangen, und der Methodenaufruf an die im Dekorator enthaltene Klasse (`decoratorComponent`) weitergeleitet.

Damit stellt sich die Klassenstruktur für die Verwendung des Dekoratormusters wie folgt dar: (siehe Abbildung 7.6):

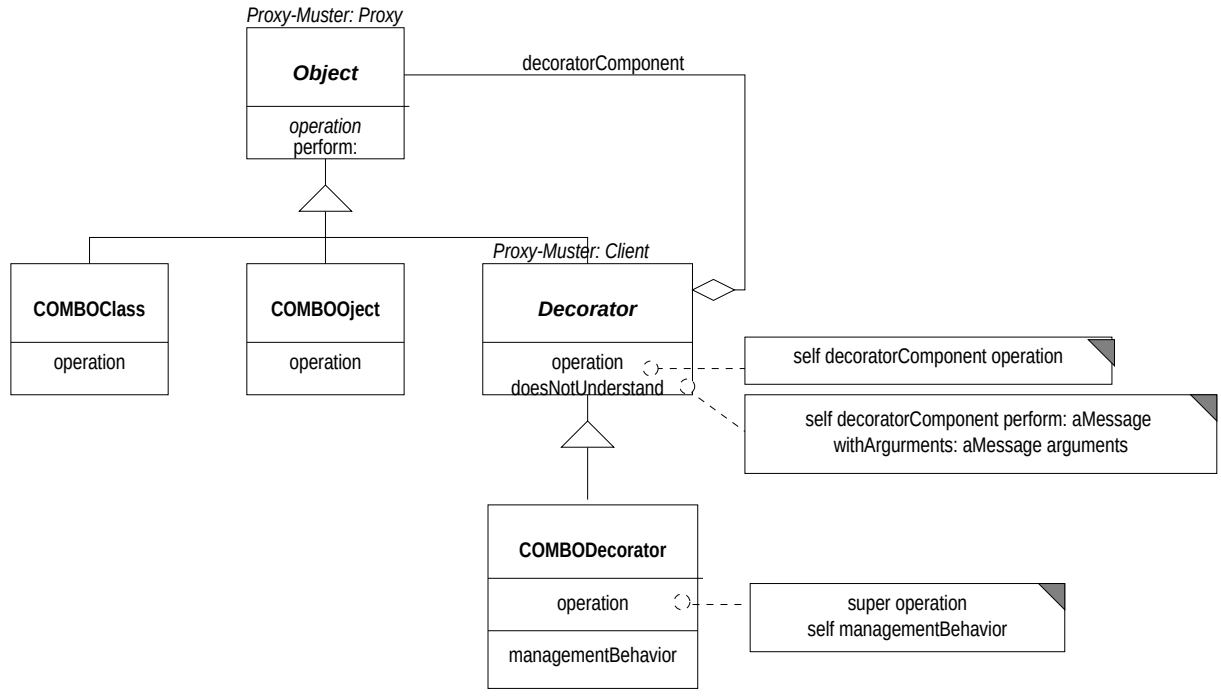


Abbildung 7.6: Verwenden der Entwurfsmuster „Dekorator" und „Proxy" in COMBO

Mit diesem Grundgerüst kann ein aktueller Entwurf als eine Sammlung von Benutzungsschnittstellen-Elementen aufgebaut werden.

Wie in Kapitel 6 dargestellt, werden solche Benutzungsschnittstellen-Elemente mithilfe verschiedener Editoren, d. h. Spezifikationswerkzeuge spezifiziert. Jeder Editor stellt die im Modell enthaltenen Elemente auf eine bestimmte Weise dar, d. h. ein Widget wird im Zeichenprogramm durch ein graphisches Benutzungsschnittstellen-Element dargestellt und im Objektdiagrammeditor durch die schematisierte Darstellung eines Objekts.

Zu den Verwaltungsaufgaben der Klasse `COMBOProjekt` gehört es daher auch, zu koordinieren, in welchen Editoren ein Element des aktuellen Entwurfs dargestellt wird. An die Editoren wird die Darstellung und die Speicherung der Darstellung delegiert (siehe Abschnitt 7.3.2).

Ein konkretes Projekt kann daher, wie in Abbildung 7.7 gezeigt, schematisch dargestellt werden: Im Modell sind die Dekoratoren gespeichert, die wiederum eine Liste der Editoren verwalten, mit denen die zugehörigen Objekte spezifiziert werden.

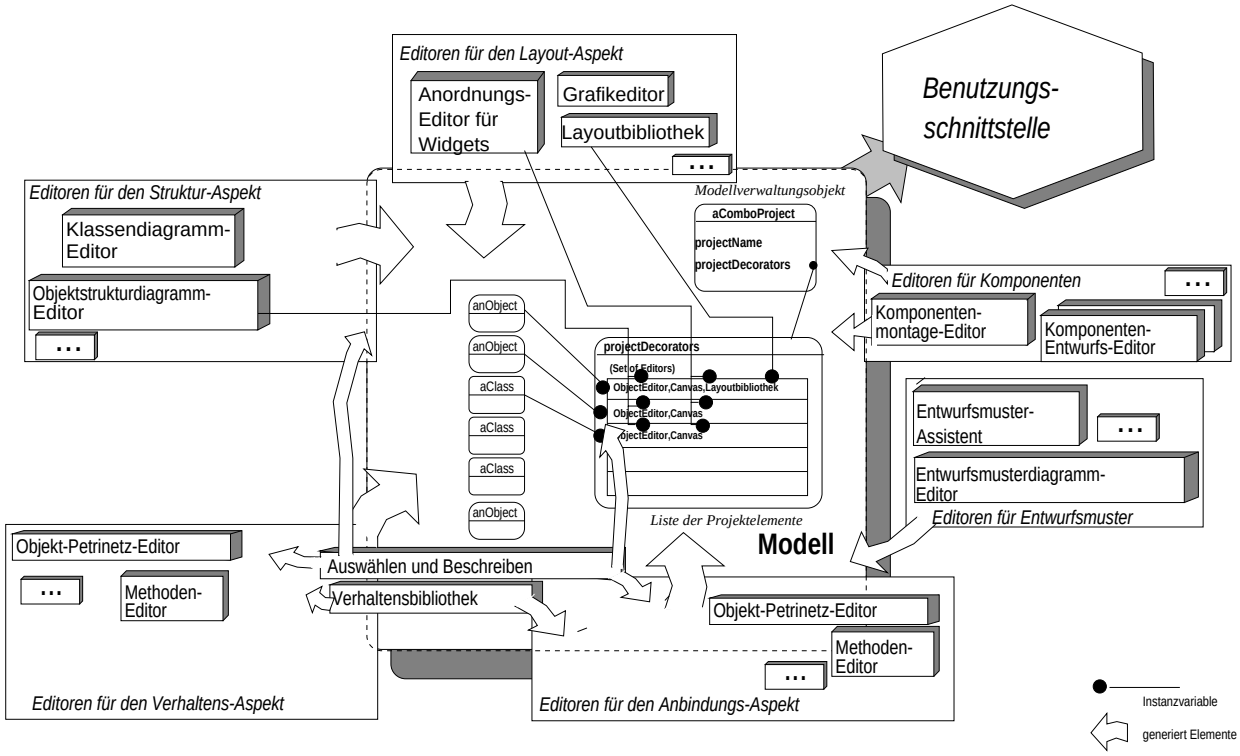


Abbildung 7.7: Das Modell und das Zusammenwirken der Editoren in einem konkreten Entwurf

7.3.2 Implementierung der Spezifikationswerkzeuge: Die COMBO-Editoren

Die verschiedenen Editoren als Realisierung von Spezifikationsmethoden wurden in Kapitel 6 beschrieben. In diesem Abschnitt wird die Integration der Editoren und der von ihnen erzeugten Elemente beschrieben.

Verschiedene Arten der Elemente von Benutzungsschnittstellen

Wie in Abschnitt 7.3.1 dargestellt, ist das COMBO-Modell flexibel genug, verschiedene Arten von Benutzungsschnittstellen-Elementen zu integrieren.

Im aktuellen Prototyp werden drei verschiedene Realisierungen der Funktionalität von Benutzungsschnittstellen unterstützt:

- Der klassische MVC-Mechanismus,
- das Application Framework von VisualWorks (beide in Abschnitt 4.3.2 vorgestellt) und
- das in Abschnitt 6.7.1 vorgestellte Komponentenmodell.

Dies ist möglich, weil die verschiedenen, auf diesen Mechanismen aufgesetzten, Editoren (MVC-Editor (Abschnitt 6.3.4), UIPainter (Abschnitt 6.1.1) und Komponenteneditor (Abschnitt 6.7.1)) Teile von Benutzungsschnittstellen erzeugen, die gemäß dem jeweiligen Prinzip für sich vollständig funktionieren.

Die COMBO-Projektverwaltung speichert die erzeugten Elemente unabhängig von ihrem Funktionalitätsprinzip. Deshalb könnten auch alle anderen in Abschnitt 4.3.2 vorgestellten Funktionalitätsprinzipien integriert werden. Wenn es sich um das gleiche Prinzip, aber eine andere Implementierung handelt, kann sogar die gleiche Spezifikationsmethode verwendet werden.

Hier wird der Vorteil eines weitgehend implementierungsunabhängigen Vorgehens durch das Aspektmodell deutlich: Unabhängig davon, daß es viele verschiedene Funktionalitätsprinzipien (von denen die wichtigsten in Abschnitt 4.3.2 beschrieben wurden) gibt, kann eine Benutzungsschnittstelle mit Hilfe der Begriffe *Aussehen, Struktur, Verhalten* und *Anbindung* beschrieben werden. Neue Funktionalitätsprinzipien wie die der Komponententechnologie können dann relativ einfach integriert werden, ohne daß sich das konzeptuelle Modell der Entwickler/in an Implementierungsdetails anpassen muß. Die Integration und Implementierung muß allerdings von der Werkzeug-Entwickler/in realisiert werden.

Verschiedene Arten von Editoren

Im wesentlichen gibt es drei Arten von Editoren, die bei Abgabe dieser Arbeit in COMBO verwendet werden: UIPainter, HotDraw-Editoren und, als Ergänzung für die textuelle Programmierung, der VisualWorks Klassenstöberer.

Neue Editoren von COMBO können entweder als Erweiterungen der vorhandenen Editoren erstellt werden oder es kann auch auf ein beliebiges neues Konzept zurückgegriffen werden.

Die Struktur und Mechanismen des VisualWorks UIPainters wurden in Abschnitt 4.3.2 vorgestellt. Beispielsweise hat Martin den UIPainter für den LCL-Komponenten-Editor (siehe Abschnitt 6.7.1) erweitert.

Die Mechanismen der HotDraw-Editoren sind in [Jac00] beschrieben. Die Struktur des Entwurfsrahmens HotDraw und die Punkte, an denen die Erweiterungen stattfinden, werden in Abbildung 7.8 zusammengefaßt.

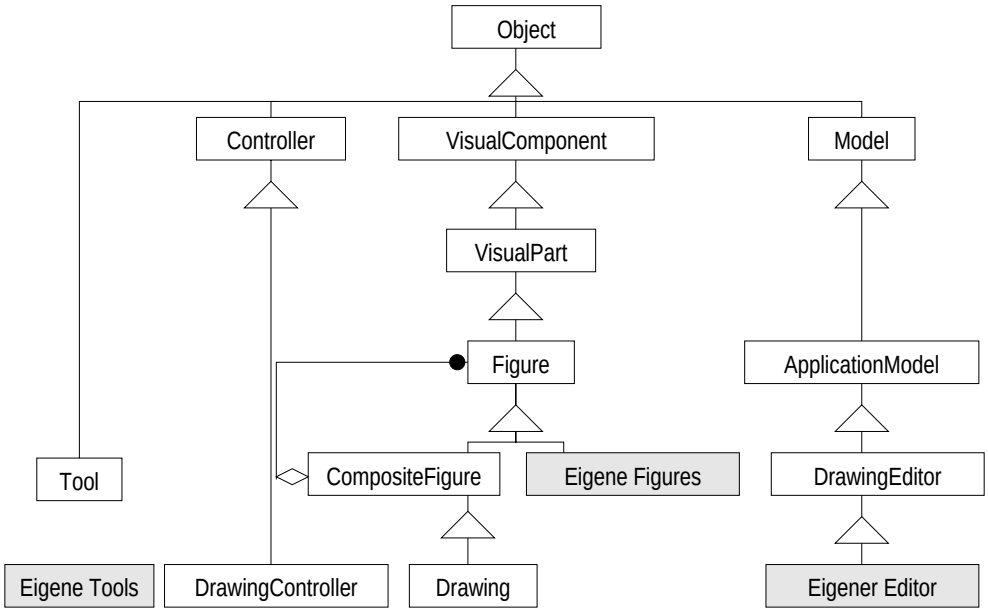


Abbildung 7.8: Die Struktur von HotDraw-Editoren

Durch die Verwendung des oben beschriebenen Dekorator-Musters in der Projektverwaltung werden die Abbildungen der Editoren konsistent gehalten.

Elemente, die mit den Zeichen- oder Komponentenwerkzeugen von HotDraw erstellt wurden werden mit Hilfe der VisualWorks-Klasse `ViewHolder` in die UIPainter-Editoren eingebunden.

7.4 Durchgängigkeit der Vorgehensweise in den verschiedenen Programmierebenen im Tank-Beispiel: Das MVC-Prinzip

Die in Abschnitt 5.3 auf Seite 133 beschriebene Vorgehensweise des vertikalen „Absteigens" von Programmierebene zu Programmierebene soll hier anhand des MVC-Prinzips im Tankbeispiel beschreiben werden.

Die Erfahrungen mit neun studentischen Arbeiten (Diplom und Studienarbeiten sowie Praktika) zeigten, daß Informatikstudent/innen vier bis sechs Wochen Zeit benötigen, um Anwendungen in VisualWorks/Smalltalk mit Hilfe des MVC-Prinzips programmieren zu können (bei täglicher Beschäftigung mit dem Thema).

Die Hälfte der Zeit wird zum Erlernen der Klassen und Mechanismen für die Benutzungsschnittstelle benötigt.

Das Vorgehen, erst die Entwicklungsumgebung VisualWorks und die Sprache Smalltalk zu erlernen und danach die textuelle Programmierung der Benutzungschnittstelle, soll hier als

Bottom-Up-Verfahren bezeichnet werden. Personen, die auf diese Weise vorgegangen sind, verstehen das Aspektmodell und die aufgabenorientierte Programmierung.

Auch das umgekehrte Vorgehen, d. h. *Top-Down* von der aufgabenorientierten zur textuellen Programmierung, sollte getestet werden. Damit sollten die Grundideen dieser Arbeit, der Erklärungsansatz und der Aspektansatz verifiziert oder verworfen werden:

Wenn die Benutzungsschnittstelle mit Hilfe der Aspekte des Aspektmodells erklärt werden kann, dann kann umgekehrt die Benutzungsschnittstelle auch aus Spezifikationen, die die Form solcher Erklärungen haben, konstruiert werden.

Der Versuch ist die Weiterführung von Untersuchung 1: „Direktmanipulatives vs. strukturorientiertes Programmieren einer Robotersteuerung" in Abschnitt 2.2.6 auf Seite 28: Statt visueller Programmierung auf den verschiedenen Programmierebenen für eine Steuerung, soll hier die visuelle Programmierung auf den verschiedenen Programmierebenen von Benutzungsschnittstellen untersucht werden.

Folgende Hypothese soll untersucht werden:

Bietet ein Werkzeug visuelle Methoden für die Spezifikation des Verhaltensaspekts sowohl *aufgabenorientiert* (Ebene 3) als auch *strukturorientiert* (Ebene 2) an, so daß ein Top-Down-Vorgehen möglich ist, ist das MVC-Prinzip schneller zu erlernen als auf dem Bottom-Up-Weg. Durch das Top-Down-Vorgehen wird auch das Verständnis von textueller Programmierung (Ebene 1 und 0) erhöht.

Aus Zeitgründen konnte kein formaler Test stattfinden. Stattdessen wurden zwei Versuchsperson ausgewählt, eine, die das Tank-Beispiel im Bottom-Up-Verfahren programmierte (und weiterhin mit BU abgeküzt wird) und eine, die das Top-Down-Verfahren erprobte (TD abgekürzt).

- Versuchsperson BU, ein Informatiker, erlernte Smalltalk in 32 Lektionen nach [Hop97]. Die MVC-Programmierung wurde anhand einer Beispiel-Benutzungsschnittstelle gelernt, die einen ausgewürfelten Zahlwert auf dem Bildschirm anzeigt. Diese Beispiel-Benutzungsschittstelle sollte den MVC-Mechanismus verwenden. Das Tankbeispiel mußte sie anhand dieses Beispiels von Hand programmieren.
- Versuchsperson TD, ein Diplom-Elektrotechniker, z.z. Zt. Informatikstudent, kannte die Programmierprinzipien von Smalltalk in Grundzügen. An ihm wurde die in Abschnitt 5.3 dargestellte Vorgehensweise erprobt.

Top-Down-Vorgehen

Tabelle 7.1 gibt die Vorgehensschritte, ihre Ausführung und die Zeit, die TD für die einzelnen Schritte benötigt hat, wieder.

Vorgehensschritt	**Ausführung**	**Zeit TD**
1. Verstehen des Aspektmodells	Das Aspektmodell wurde mündlich erklärt.	25 min
2. Visuelle Spezifikation mit aufgabenbezogenen Methoden (Ebene 3)	Zunächst wurde das Würfelbeispiel mit der Verhaltensbibliothek vorgeführt, dann sollte TD einen Tank ebenfalls mit der Verhaltensbibliothek entwerfen.	30 min
3. Wenn diese nicht ausreichen:		
3.a. Visuelle Spezifikation mit Komponenten (Abstieg Ebene 2)	wurde nicht erprobt	-
3.b. visuelle Spezifikation mit Hilfe von Entwurfsmustern (ebenfalls Ebene 2) oder	Mithilfe des MVC-Editors, der in Abschnitt 6.3.4 beschrieben wurde, wurde das Würfelbeispiel auf Ebene 2 vorgestellt. Danach sollte anhand dieses Beispiels der Tank auf Ebene 2 programmiert werden	55 min
3.c. visuelle Spezifikation sonstiger Mechanismen (Ebene 2)	wurde nicht erprobt	-
4. Wenn diese nicht ausreichen: Textuelle Spezifikation mit Entwurfsmustern (Abstieg auf Ebene 1)	TD sollte Ausschnitte aus der Klassenbibliothek mit Hilfe des textuellen Klassenstöberers erklären. Damit sollte geklärt werden, ob sie das MVC-Prinzip verstanden hatte.	10 min
5. Normale textuelle Programmierung (Abstieg auf Ebene 0)	wurde nicht erprobt	-

Tabelle 7.1: Das Top-Down Vorgehen nach dem Aspekt- und Erklärungs-Ansatz Vorgehensschritte, Ausführung und benötigte Zeit im konkreten Versuch

Die Zeiten zeigen bereits, daß TD schnell gelernt hat und das Beispiel schnell entwickeln konnte. Zur Überprüfung sollte TD einer anderen Person das Beispiel erklären, indem es sie noch einmal entwickelt. Abbildungen 7.9 bis 7.11 zeigen Aussschnitte aus dem Entwicklungsprozeß.

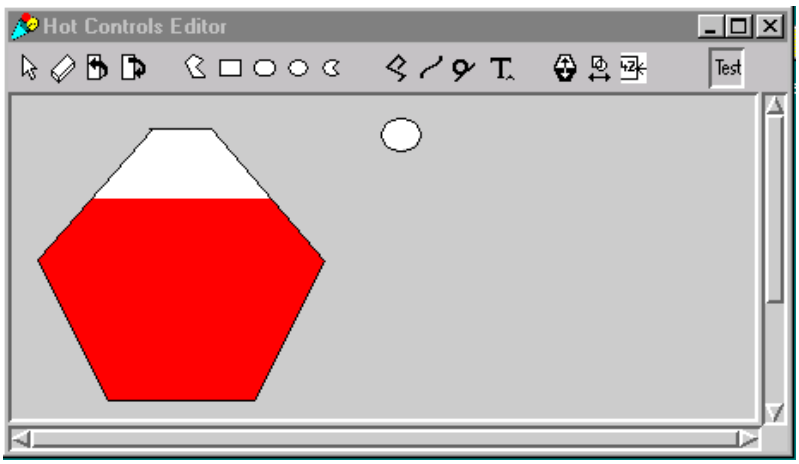


Abbildung 7.9: Aufgabenorientierte Programmierung (Ebene 3) durch Verhaltensbibliothek

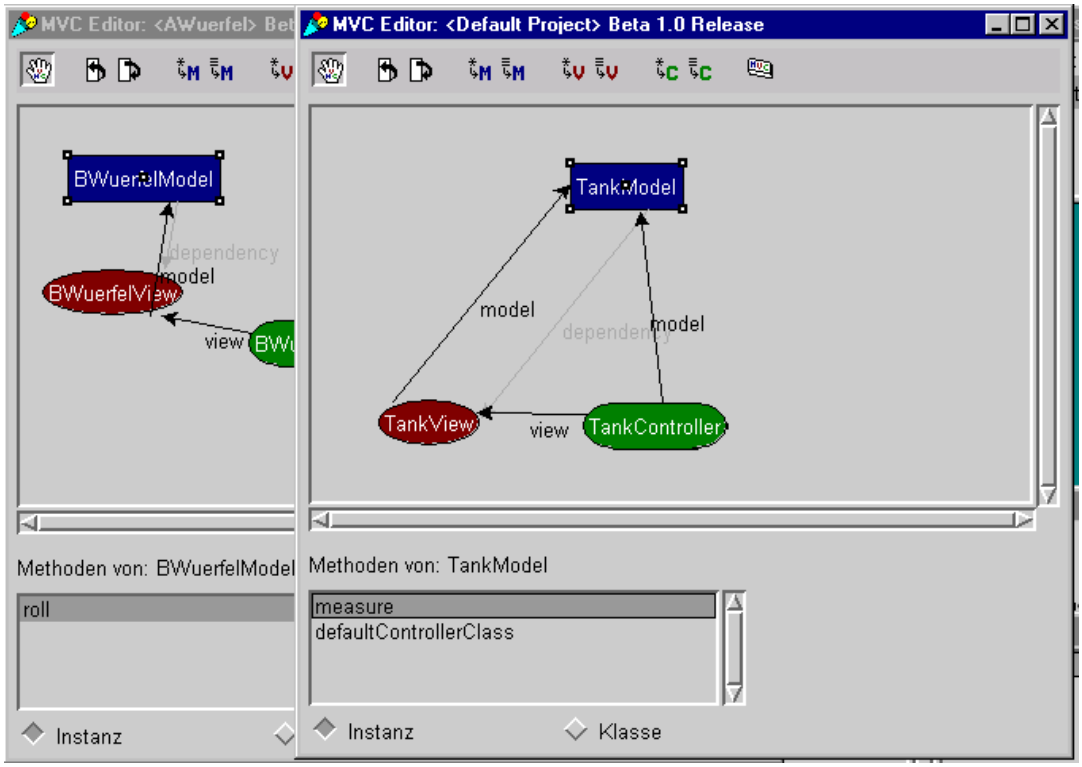


Abbildung 7.10: MVC-mechanismusorientierte Programmierung (Ebene 2). Die Würfel-Anwendung dient als Beispiel

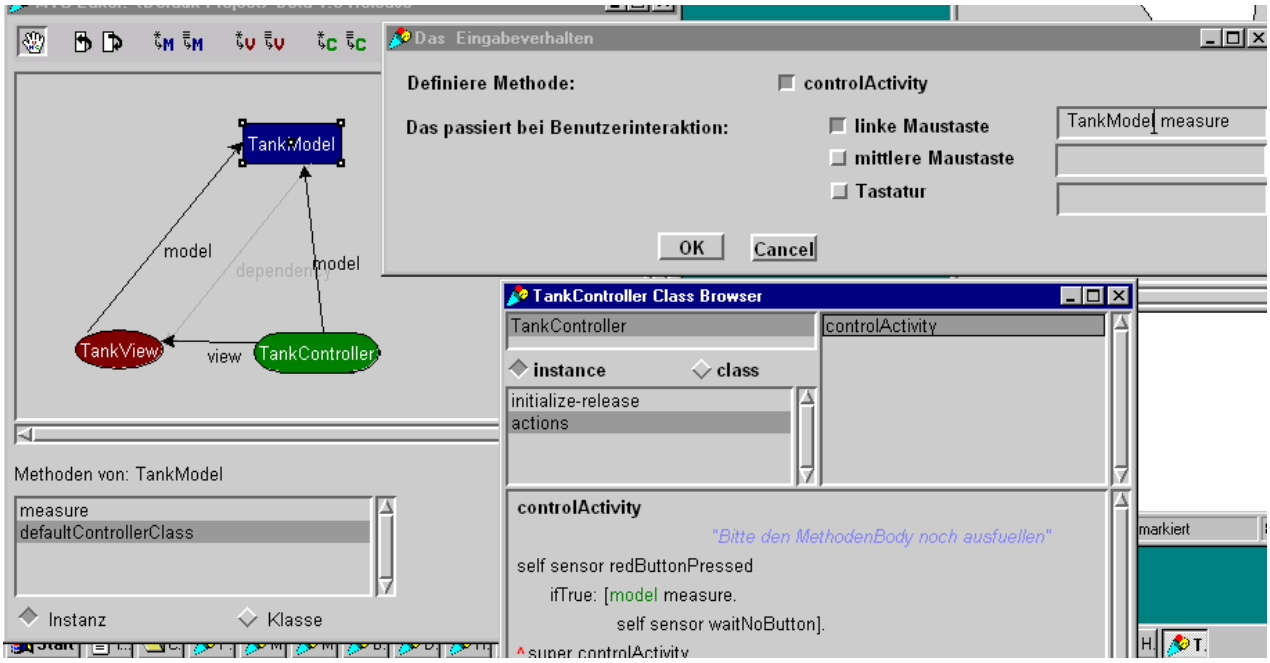


Abbildung 7.11: MVC-Mechanismusorientierte Programmierung (Ebene 2). Der Übergang zur textuellen Programmierung auf Ebene 1 und 0 wird durch den Methodenstöberer übergeleitet.

Zusätzlich beantwortete TD Fragen zu Grundideen, Anwendung und Programmierung des

MVC-Mechanismus richtig. Auch fand sie sich in der textuellen Programmierung (im Klassenstöberer) gut zurecht, da sie herleiten konnte, welche Klassen, Methoden und Programmtext-Stücke warum generiert wurden. Die Funktionsweise wurde in der Terminologie des Aspektmodells erkärt.

Damit kann für diesen Fall die Hypothese verifiziert werden.

Eine Anmerkung: Interessanterweise stellte TD nach Abschluß des Versuchs die Frage: „Was ist denn jetzt an dieser Vorgehensweise das Besondere? Das ist doch eigentlich ganz banal". Offensichtlich hat TD durch die Vorgehensweise die bei den Student/innen beobachteten Schwierigkeiten beim Erlernen der Programmierung von Anwendungen mit dem MVC-Mechanismus nicht gehabt.

Bottom-Up-Vorgehen

Die folgende Tabelle gibt die Zeiten für BU wieder:

Vorgehensschritt	**Zeit**
Erlernen der Grundlagen von Smalltalk	49 h
Erlernen der Graphikprogrammierung mit VisualWorks	3,5 h
Erlernen der Klassen und Mechanismen der VisualWorks Klassenbibliothek für Benutzungsschnittstellen	13 h
Programmieren des Tank-Beispiels durch Modifikation eines Beispiels im Buch	4h

Tabelle 7.2: Zeiten zum Erlernen des Programmierens des Tank-Beispiels „von Hand"

Es ist deutlich zu sehen, daß der Aufwand zum Erlernen der Grundlagen für die Programmierung des Tank-Beispiels sehr hoch ist. Trotzdem fühlt sich BU noch nicht sehr sicher in der Smalltalk-Programmierung. Beispielsweise schreibt die Versuchsperson nach drei Wochen:

> *„Ich bin jetzt mit dem Buch bis zum Wuerfel gekommen. Der Wuerfel rollt auch bischen. Leider stecke ich gerade fest und hab' gerade keine richtige Lust mehr mich durchzubeissen. Vielleicht ist es hilfreich, wenn wir uns mal zusammensetzen."*

Das MVC-Prinzip erklärt sie mit Hilfe ähnlicher Zeichnungen wie TD.

Vergleich

Der Vergleich ist nicht repräsentativ, denn

- TD hatte von vorneherein für den Versuch nur einen Nachmittag Zeit, während BU Smalltalk von Grund auf erlernen wollte und sich vier Wochen Zeit nahm,
- da die Autorin dieser Arbeit auch die Versuchsleiterin war, ist der Versuch nicht objektiv,
- die prototypische Realisierung von COMBO ist nicht sehr robust (während des Versuchs gab es einen Absturz) und komfortabel.
- Der Versuch zeigt aber das Vorgehen zweier an der Entwicklung von COMBO nicht beteiligten Personen und hat die Hypothese bestätigt.

7.5 Anwendbarkeit von COMBO

In der Einleitung wurden als Beispiele der Tank und auf Seite 5 die komplexeren Flugzeug-Benutzungsschnittstellen vorgestellt. Der Tank wurde durchgängig in der ganzen Arbeit als Beispiel für die einzelnen Methoden verwendet, d. h. es ist einsichtig, daß mit COMBO ein solches Tank-Widget erstellt werden kann.

Wie ist es aber mit komplexeren Benutzungsschnittstellen?

Benutzungsschnittstellen für Flugzeuganzeigen

Die klassische Flugzeuganzeige

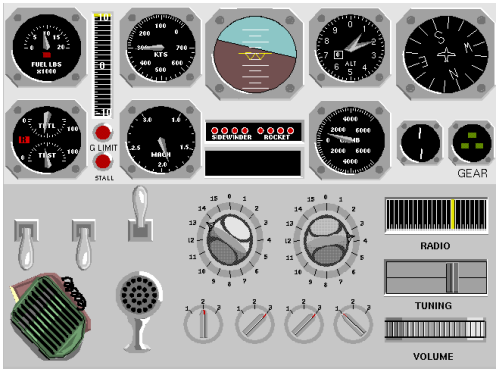
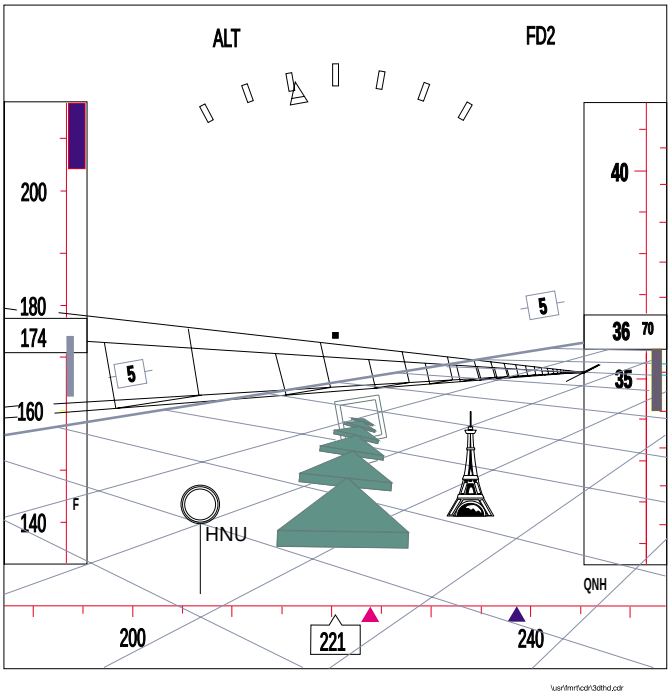


Abbildung 7.12: Die klassische Flugzeug-Benutzungsschnittstelle

Abgesehen davon, daß diese Widgets von der Firma Genlogic [Gen00] bereits vordefiniert wurden, wäre zur Erstellung der klassischen Flugzeuganzeigen-Benutzungsschnittstelle folgendes Vorgehen möglich:

- Vorhandene Widgets werden mit Hilfe des Anordnungswerkzeugs auf einer Arbeitsfläche plaziert.
- Noch nicht definierte Widgets werden
  - mit Hilfe des Zeicheneditors gezeichnet,
  - ihr Verhalten wird mit Hilfe der Verhaltensbibliothek (z. B. bei den Skalen) oder des MVCEditors definiert und
  - ihre Anbindung an die Objekte zur Verwaltung der Flugdaten wird durch die Verbindung der entsprechenden Attribute (und Formeln zur Berechnung) mit den Widgets definiert.
- Danach können auch diese Widgets auf der Arbeitsfläche plaziert werden, und nötige Anbindungen vorgenommen werden.



Für die Skalen kann hier, ähnlich wie im ersten Beispiel, vorgegangen werden: Spezifizieren durch Zeichnen und Zuweisen von Verhalten.
Die Dreiecke in der Mitte der Anzeige können ebenfalls gezeichnet werden. Ihr Verhalten (Änderung von Größe und Position in Abhängigkeit von Fluggeschwindigkeit und Lage) ist in der prototypischen Implementierung von COMBO nicht vorhanden und müßte neu implementiert und der Verhaltensbibliothek hinzugefügt werden.

Abbildung 7.13: Skizze des Vorgehens zum Spezifizieren einer experimentellen Flugzeug-Benutzungsschnittstelle

Der Hintergrund, d. h. ein sich veränderndes Koordinatensystem oder eine Kamerabild bzw. ein simuliertes Kamerabild, muß von Hand programmiert werden. Um hierfür eine visuelle Spezifikationsmethode zu entwickeln, müßten die Algorithmen der Computer-Graphik nach entsprechenden Mustern abgesucht werden. Würden solche Muster gefunden, müßte überprüft werden, ob eine Werkzeugunterstützung möglich ist, d. h. ob der Code zur Implementierung der Muster effizient aus einer visuellen Spezifikationsmethode generiert werden könnte.

7.6 Erweiterungen für COMBO

7.6.1 Verteiltes Entwerfen von Benutzungsschnittstellen

In [SBF96] wurde ein *erweiterter Model View Controller-Mechanismus für Gruppenarbeit* vorgestellt. Dabei wird ein erweitertes COMBO-Modell benötigt, das außer der vielfachen Darstellung der Elemente der Benutzungsschnittstelle in den Editoren auch den Zugriff mehrerer Entwickler/innen verwaltet. Der Zugriff selbst wird von Agenten gesteuert, die sich miteinander koordinieren können.

Das Grundmodell ist in Abbildung 7.14 dargestellt, es wurde jedoch nicht implementiert.

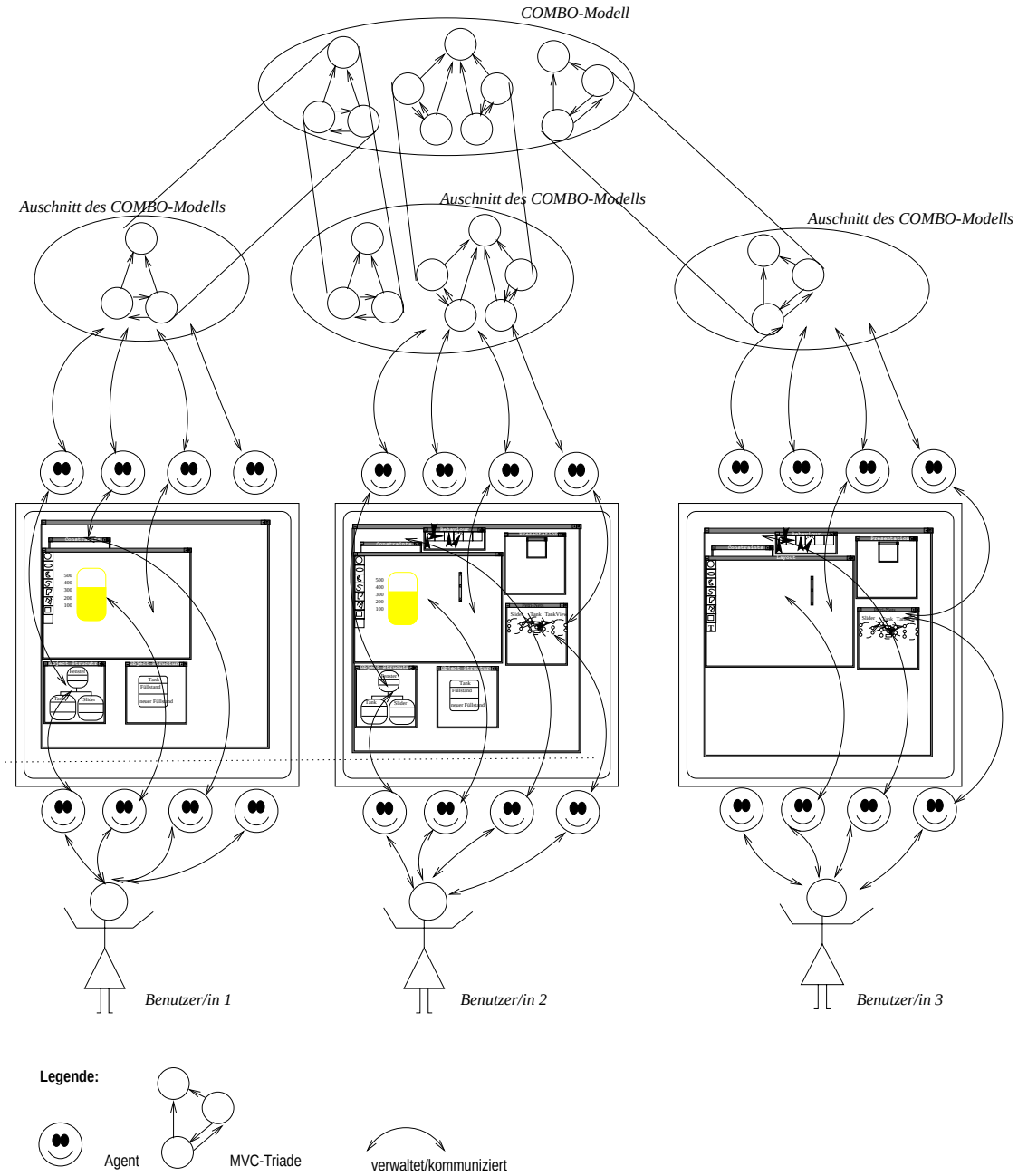


Abbildung 7.14: Grundmodell für verteiltes Entwickeln mit COMBO

7.6.2 Erweiterung von COMBO zur Entwicklung beliebiger Benutzungsschnittstellen

COMBO wurde als Werkzeug zur Entwicklung von Benutzungsschnittstellen für technische Systeme entwickelt, weil dort dynamische Benutzungsschnittstellen-Elemente benötigt werden.

Auch Schreibtisch-Benutzungsschnittstellen verwenden dynamische Elemente, wie z. B. das Papierkorb-Piktogramm von Microsoft Windows, je nach Zustand, voll oder leer dargestellt wird:

> *"In the future, visual representations will resemble real-world objects, which act like real-world objects while the user interacts with them. Today, icons are static bitmaps that do little to convey the capabilities of an object"* [IBM97b]

Die Benutzungsschnittstellen, die mit COMBO erstellt werden können, sind zweidimensional und basieren auf Maus- und Tastaturinteraktion.

Um weitere Eingabegeräte einbinden zu können und multimodale oder dreidimensionale Benutzungsschnittstellen der künstlichen oder erweiterten Realität entwickeln zu können, könnten die Ideen von COMBO verwendet werden. Allerdings müßte dreidimensionale Graphik und Interaktion von der Programmierumgebung unterstützt werden oder ein entsprechender Entwurfsrahmen speziell für COMBO entwickelt werden.

Außerdem müßten neue Spezifikationsmethoden entwickelt werden, die der Entwicklung dreidimensionaler Benutzungsschnittstellen angepaßt würden. Die in Abschnitt 2.3 vorgestellten Prinzipien für die Unterstützung visueller Spezifikationsmethoden, sowie die Basisarchitektur von COMBO, wären aber eine gute Grundlage.

Die Vorgehensweise, die für COMBO verwendet wurde, ist auch für solche Werzeuge anwendbar:

- Herausfinden, wie Entwickler/innen sich die Benutzungsschnittstellen vorstellen,
- aus den Ergebnissen die übergreifende Terminologie herausfiltern und
- auf dieser Terminologie aufbauend Spezifikationsmethoden entwickeln.

7.6.3 Erweiterung von COMBO durch neue Methoden und Prinzipien

Die offene Realisierung erlaubt das Einbinden neuer Methoden und die Erweiterung durch neue Prinzipien.

Zum Einbinden neuer Methoden müßten neue Editoren geschrieben werden, die in die COMBO-Verwaltung integriert werden müssen. Da die Darstellung den Editoren überlassen bleibt, und das Modell, wie oben beschrieben, nur enthält, welche Editoren benachrichtigt werden, ist ein solches Einbinden neuer Methoden nicht sehr schwer.

Schwieriger ist das Finden neuer Methoden, die das konzeptuelle Modell nicht zu sehr überladen und einfach zu verwenden sind.

Mögliche Methoden wären:

Die **unmittelbare „Papier- und Stift-Interaktion"**, die in Kapitel 1 vorgestellt wurde, ist eine interessante Erweiterung. Solche Interaktionen machen die Entwicklung noch intuitiver.

Das **Verwenden von Constraints** wären eine gute Erweiterung für COMBO, sowohl, weil sich Zusammenhänge im graphischen Verhalten oder bei der Anbindung leichter als Programmcode formulieren ließen, als auch, weil es viele visuelle Ansätze zur Constraintprogrammierung gibt. In COMBO wurden sie bisher nicht aufgenommen, weil es bereits viele visuelle Ansätze für Constraints gibt, und aus Forschungssicht keine neuen visuellen Spezifikationsmethoden dabei entstanden wären.

7.6.4 Erweiterung von COMBO für ergonomische Benutzungsschnittstellen

Mit COMBO können Benutzungsschnittstellen leicht entworfen werden. Damit ist aber nicht sichergestellt, daß die entworfenen Benutzungsschnittstellen auch einfach zu benutzen sind. Ein wesentlicher Schritt dazu ist die Integration von Werkzeugen, die z. B. das Einhalten von Richtlinien überwachen. Dilli [DH94] hat eine Komponente für das Werkzeug DIADES II [DVH93] entworfen, die die Integrität von Benutzbarkeitsregeln überwacht. Eine solche Komponente könnte ebenfalls in COMBO integriert werden. Dazu gibt es zwei Ansätze:

- Die Integritäts-Komponente überwacht den aktuellen Entwurf, der im COMBO-Modell gespeichert ist.
- Die Editoren integrieren die für sie infrage kommenden Richtlinien. Ansatzweise ist dies in COMBO durch den Einsatz von Entwurfsmustersprachen bereits der Fall, z. B. bei den anwendungsspezifischen Layoutmustern (siehe Abschnitte 4.4.2 und 6.1.3)

7.6.5 Evaluierung von COMBO

Das Experiment in Abschnitt 7.4 testete nur einen kleinen Ausschnitt der Entwicklungsmethodik mit COMBO. Die Bewertung aller visuellen Methoden einzeln und des Werkzeugs COMBO als Ganzes, mit Hilfe des in Kapitel 2.2 eingeführten kognitiven Rahmens, durch ein Experiment mit Entwickler/innen, steht noch aus. Ein Experiment wäre jedoch sehr aufwendig, wie im folgenden in Grundzügen dargestellt:

Dazu müßte ein Beispiel, das so komplex ist, daß alle Aspekte definiert werden müßten, von vielen Entwickler/innen einmal „von Hand" d. h. textuell auf Ebene 0, und einmal mit Hilfe von COMBO entwickelt werden. Um eine aussagekräftige Bewertung zu erhalten, müßten so viele Testpersonen daran teilnehmen, daß statistisch signifikante Ergebnisse entstehen, und zwar über verschiedene Gruppen:

- Ingenieur/innen und Informatiker/innen,
- Personen, die eher visuell orientiert sind und Personen, die eher textuell orientiert sind,
- Gruppen, die bestimmte Kombinationen der angebotenen Methoden nutzen und
- Gruppen, die die Grundkonzepte von objektorientierten Benutzungsschnittstellen und Software Engineering Methoden kennen bzw. nicht kennen.

Es fanden sich am Fachgebiet neben den an der Entwicklung von COMBO beteiligten Personen (deren Ergebnisse auch verfälscht wären) nicht genug Personen, die das notwendige Wissen über Benutzungsschnittstellen-Entwicklung mit Smalltalk besaßen, um an diesem aufwendigen Experiment teilnehmen zu können.

Kapitel 8

Zusammenfassung und Ausblick

Mit der vorliegenden Arbeit werden ein Modell, eine Vorgehensweise und ein Werkzeug zur Entwicklung von Benutzungsschnittstellen technischer Systeme vorgelegt.

Zunächst wurden in allen Disziplinen die *Anforderungen* herausgearbeitet, die *an Methoden und Werkzeuge zur Entwicklung von Benutzungsschnittstellen* gestellt werden:

- Aus der *Literatur* der Fachrichtungen „kognitive Psychologie", „Software-Ergonomie" und „visuelle Programmierung" sowie den *Ergebnissen einer Untersuchung* zum Thema „Visuelles Programmieren von Robotern" wurden die *kognitiven Anforderungen* extrahiert, die *an* die *Menschen beim (visuellen) Entwickeln von Software* gestellt werden, und die für eine Werkzeugunterstützung aufgegriffen werden müssen:

  - Konzeptuelle Modelle der Entwickler/innen bilden sich aus dem Wissen über die zukünftigen Benutzer/innen, den technischen Modellen und Vorgehensweisen bei der Software-Entwicklung von Benutzungsschnittstellen. Ein Werkzeug zur Entwicklung von Benutzungsschnittstellen muß daher den Aufbau dieser konzeptuellen Modelle unterstützen. Dazu wurde in Abschnitt 2.1.3 der *Erklärungsansatz* erarbeitet. Ebenso muß ein Werkzeug eine Terminologie anbieten, die das konzeptuelle Modell berücksichtigt. Dazu wurde in Abschnitt 2.1.4 der *Aspektansatz* vorgestellt.
  - Programmierung findet auf verschiedenen Abstraktionsebenen statt. Das hat zur Folge, daß es in einer Lernkurve immer wieder „Mauern" gibt, die die Entwickler/in beim Erlernen der Software-Entwicklung von Benutzungsschnittstellen überwinden muß. Für die visuelle Spezifikation von Benutzungsschnittstellen wurden in dieser Arbeit *vier Ebenen der Programmierung* benannt: die Ebene der Basiskonzepte (Ebene 0), die Struktur- und Mechanismenebene, die übergeordnete Konzepte textuell (d. h. mit Hilfe von Programmtext) beschreibt (Ebene 1), die Struktur- und Mechanismenebene, die übergeordnete Konzepte visuell beschreibt (Ebene 2) und die Ebene des aufgabenorientierten Programmierens (Ebene 3).
  - Zur Unterstützung des Erklärungs- und des Aspektansatzes sowie zur Erleichterung des Programmierens auf den vier Abstraktionsebenen wurden in dieser Arbeit fünf Interaktionsprinzipien für Benutzungsschnittstellen-Werkzeuge aufgestellt.

- Aus einer *Umfrage* unter Ingenieur/innen und Informatiker/innen und der *Analyse von Benutzungsschnittstellen technischer Anwendungen* sowie der *Literatur* technischer Fachrichtungen wurden *Besonderheiten beim Entwurf von Benutzungsschnittstellen technischer Anwendungen* herauskristallisiert:

  - Benutzungsschnittstellen technischer Anforderungen sind von der Graphik her eher dynamisch, von der Anordnung her eher statisch. Prozeßbilder spiegeln den Prozeß und die Anlage wider. Entwicklungs-Werkzeuge müssen den Entwurf solcher Benutzungsschnittstellen vereinfachen.
  - Die Darstellung der Benutzungsschnittstellen muß verschiedene Betriebsarten unterstützen und darf die Bediener/in nicht überfordern. Solche graphischen Darstellungen erfordern neue, ungewöhnliche graphische Benutzungsschnittstellen-Elemente, deren Erstellung durch ein Werkzeug unterstützt werden sollte.
  - Entwickler/innengruppen für den Entwurf solcher Benutzungsschnittstellen sind interdisziplinär zusammengesetzt, deshalb muß die Verständigung zwischen verschiedenen Fachrichtungen unterstützt werden.
  - Die Expert/innen der technischen Anwendungsgebiete sind ausgebildet, Systeme, z. B. Fabrikanlagen oder Anlagensteuerungen, zu entwerfen. Oft wollen sie auch die Benutzungsschnittstellen zu den Programmen erstellen, möglichst mit den Methoden, mit denen sie ausgebildet wurden. Deshalb müssen Werkzeuge, die von Anwendungsexpert/innen genutzt werden sollen, sowohl die Terminologie als auch die Methoden der technischen Fachgebiete unterstützen.

- Auch die *software-technischen Grundlagen* zur Entwicklung von Benutzungsschnittstellen wurden herausgearbeitet. Da in der Informatik-Literatur implementierungs- und methodenunabhängige Beschreibungen des Erstellens von Software für Benutzungsschnittstellen fehlen, wurde mit Kapitel 4 ein Überblick gegeben. Dieser Überblick gibt den *Stand der Literatur*, die *Ergebnisse einer Umfrage im WWW* und die im Rahmen dieser Arbeit gemachten *Erfahrungen mit* vielen *Programmierumgebungen* und einigen -sprachen wieder. Dazu gehören

  - die *Terminologie* von Benutzungsschnittstellen sowie ihr *grundsätzlicher Aufbau*,
  - verschiedene *Modularisierungsmodelle* von Benutzungsschnittstellen und die Aufteilung ihrer Aufgaben,
  - *Modelle über Funktionsweisen* von Benutzungsschnittstellen, sowie die verschiedenartigen Umsetzungen in einzelnen Programmierumgebungen durch die jeweiligen Mechanismen der Klassenbibliothek,
  - der *Einsatz von Konzepten* des Software Engineering *zur Wiederverwendung*, wie Anwendungsrahmen, Entwurfsmuster und Komponenten und
  - die *Integration von visuellen Spezifikationsmethoden* in Programmierumgebungen.

Die in den Kapiteln 2 bis 4 gewonnenen Erkenntnisse wurden in ein Modell umgesetzt: das *Aspektmodell*. Das Aspektmodell baut auf dem Aspektansatz auf und beschreibt Benutzungsschnittstellen in einer Terminologie, die Entwickler/innen aller Fachrichtungen verstehen, da es sprachunabhängig, implementierungsunabhängig und plattformunabhängig ist. Es wurde

an einem Beispiel in einer konkreten Programmiersprache (Smalltalk) gezeigt, daß das Aspektmodell die konkreten Konzepte umschreibt. Aus einer Spezifikation, die in der Terminologie des Aspektmodells erstellt ist, kann ein konkreter Programmtext generiert werden. Darüberhinaus ist das Aspektmodell auf allen identifizierten Programmier-Ebenen gültig, so daß das Wechseln zwischen den Programmierebenen erleichtert wird.

Weiterhin wurde eine Werkzeugunterstützung für das Aspektmodell diskutiert und als das Werkzeug COMBO implementiert. Dazu wurden zunächst visuelle Spezifikationsmethoden für die Aspekte vorgestellt, auch auf den verschiedenen Programmierebenen, die im Rahmen dieser Arbeit entstanden sind. Auch einige bekannte visuelle Spezifikationsmethoden wurden in dieser Arbeit an das Aspektmodell angepaßt.

Insbesondere wurden visuelle Spezifikationsmethoden für den Einsatz von Entwurfsmustern (sowohl die in dieser Arbeit beschriebenen als auch bekannte Entwurfsmuster) und die Komponententechnologie entwickelt.

Alle Spezifikationsmethoden wurden als graphische Editoren implementiert bzw. die bestehenden diskutierten Spezifikationsmethoden wurden angepaßt. Die Integration weiterer graphischer Editoren ist möglich.

Zur Integration der unterschiedlichen visuellen Spezifikationsmethoden wurde eine Datenstruktur, das sog. COMBO-Modell entwickelt, das den aktuellen Entwurf enthält und die verschiedenen Darstellungen in den Spezifikationseditoren koordiniert. Dazu dient auch die COMBO-Projektverwaltung.

Weiterhin wurden Hilfsmittel zur Navigation für die Entwickler/innen implementiert, um die Verwendung unterschiedlicher Spezifikationseditoren zu erleichtern.

Die Durchgängigkeit durch die verschiedenen Programmierebenen wurde mit sehr guten Ergebnissen an einer Versuchsperson getestet und mit den Ergebnissen einer zweiten Person verglichen, die rein textuell programmierte.

Skizzen für mögliche Erweiterungen, die zum Druckzeitpunkt noch nicht implementiert sind, wurden anschließend vorgestellt.

Ausblick

Die grundlegenden Erkenntnisse für eine fachrichtungsunabhängige Werkzeugunterstützung wurden durch diese Arbeit gewonnen und ein Demonstrator, das Werkzeug COMBO, wurde implementiert.

Diese Ergebnisse können in verschiedene Richtungen weiterentwickelt werden:

- Das Aspektmodell kann für multimodale, dreidimensionale und sonstige neue Benutzungsschnittstellen-Technologien umgesetzt werden.
- COMBO kann als kommerzielles Werkzeug weiterentwickelt werden.
- Die in Abschnitt 7.6 vorgestellten Erweiterungsskizzen sollten implementiert werden.

Darüberhinaus sollten meiner Meinung nach weiterhin grundlegende Forschungen über das konzeptuelle Modell von Software-Entwickler/innen stattfinden. Es ist notwendig, daß auch die Vorgehensweisen und Methoden großer Entwickler/innen-Gruppen bei der Entwicklung

komplexer Software-Projekte über längere Zeiträume mit den Mitteln der kognitiven Psychologie analysiert werden.

Die Entwicklung neuer Technologien für den Entwurf von Benutzungsschnittstellen bzw. Software allgemein sollte nicht das Feld von „Technologie-Gurus" und „Softwaretechnologie-Bastlern" sein, sondern auch die Erkenntnisse über „Durchschnitts-Entwickler/innen" und ihre Vorstellungen einbeziehen. So, wie in neuere Programmiersprachen (z. B. Java) die guten Technologien aus etablierten Programmiersprachen (z. B. Smalltalk) einfließen, sollten auch bewährte Technologien zur Software-Entwicklung verwendet werden. Dazu muß man sich, wie in dieser Arbeit gezeigt, in verschiedensten Gebieten umschauen, z. B. bei der Stöberertechnik der Smalltalk-Entwicklungsumgebungen, bei den visuellen Techniken aus Dokumenten-Erstellungswerkzeugen (wie HTML, PowerPoint, usw.) oder bei den Systementwicklungs-Werkzeugen der Anwendungsgebiete (wie LabViews, MATLAB, usw.).

Glossar

Anordnungswerkzeug (engl. *interface builder*) Ein Werkzeug, mit dem graphische Benutzungsschnittstellen-Elemente in ihrer graphischen Darstellung auf einer Arbeitsfläche angeordnet werden können.

Anwendung Alle Komponenten des Laufzeitsystems, die die eigentliche funktionale Verarbeitung des Systems, also z. B. ohne Ein- und Ausgabe, an die Benutzer/in übernehmen. Ein- und Ausgabe von Daten an andere Komponenten, z. B. Hardware gehören auch zur Anwendung.

Anwendungsobjekte (*business objects*) sind die Objekte, die die Anwendung modellieren, also die reine Datenverarbeitung. Synonyme: **Modell** (ist aber mehrdeutig, da in verschiedenen Kontexten verwendet), **Geschäftsobjekte** ([Kra97])

Anwendungsrahmen (*framework*) Eine Menge von Klassen, die eine Anwendung realisieren

Aspekte von Benutzungsschnittstellen *Aspekt = [...] die Art der Betrachtung von etwas [Mül85].* Benutzungsschnittstellen können unter verschiedenen Gesichtspunkten betrachtet werden, z. B. dem Layoutaspekt oder dem Verhaltensaspekt. Miteinander verwoben ergeben die verschiedenen Aspekte die ganze Schnittstelle. Der Ansatz dieser Arbeit ist, verschiedene Aspekte einer Benutzungsschnittstelle zu identifizieren, Werkzeuge für die Spezifikation der verschiedenen Aspekte zur Verfügung zu stellen und die Ergebnisse der Spezifikationen zu der Benutzungsschnittstelle zusammenzustellen. Der Begriff Aspekt ist in dieser Arbeit jedoch abstrakter verwendet als im *aspektorientierten Programmieren.*

Aspektorientiertes Programmieren Programmiermethodik, bei der bestimmte Aspekte eines Programms, die nicht in einem Element (Objekt, Prozedur) definiert werden können, sondern in vielen Elementen implementiert werden müssen (z. B. die Fehlerbehandlung) explizit definiert und dann mit Hilfe des sog. Aspektwebers auf die Elemente verteilt werden [KLM+97].

Benutzungsschnittstelle Der Begriff Benutzungsschnittstelle umfaßt alle Laufzeitkomponenten eines Systems, die dafür sorgen, daß Ausgaben der *Anwendung* an die Benutzer/innen weitergegeben werden können, und daß Eingaben von Benutzer/innen entgegengenommen und an die *Anwendung* weitergegeben werden. Im Rahmen dieser Arbeit bezeichne ich damit je nach Kontext aber auch den Programmtext, der diese Komponenten des Laufzeitsystems erzeugt.

Componentware Software, die aus Komponenten gemacht wird.

Diagramm Graphische Repräsentation einer Menge von Elementen, oft als Graph dargestellt.

Entwurfsmuster (engl. *design patterns*)

Ereignis „Aktionen oder Zustandsänderungen, die häufig vom Benutzer ausgelöst werden, auf die ein Programm antworten kann. Typische Ereignisse sind z. B. das Drücken einer Taste, das Klicken auf Schaltflächen sowie Mausbewegungen." [Uni98]

ereignisgesteuert Software, die auf äußere Ereignisse - z. B. auf einen Tastendruck oder einen Mausklick - reagiert, wird als ereignisgesteuerte Software bezeichnet. Bei ereignisgesteuerten Eingabemasken ist es z. B. nicht erforderlich, die Eingabe in einer festgelegten Reihenfolge vorzunehmen. Es können nämlich die gewünschten Felder z. B. durch einen Mausklick aktiviert werden [Uni98].

ereignisgesteuerte Programmierung Ein Programmierkonzept, bei dem ein Programm ständig eine Menge von Ereignissen prüft und entsprechend darauf antwortet, z. B. auf das Drücken einer Taste oder auf Mausbewegungen. Insbesondere der Apple Mackintosh ist bekannt dafür, daß die meisten Programme eine ereignisgesteuerte Programmierung erfordern, obwohl graphische Benutzungsschnittstellen, z. B. Microsoft Windows oder das X Window System, ebenfalls nach dieser Methode arbeiten [Uni98].

Framework siehe Anwendungsrahmen

Higher-Level-Concept Für diesen Begriff gibt es keine direkte Übersetzung. Das englische Wort *concept* bedeutet Vorstellung oder Begriff, das deutsche Wort Konzept beschreibt dagegen einen Plan oder einen ersten Entwurf. Higher-level heißt höherwertig oder „auf einer höheren Ebene/Stufe". Im Deutschen muß aber noch die Dimension, auf die sich das höher bezieht, angegeben werden. In der Regel ist mit dem Begriff gemeint, daß bestimmt Mechanismen gekapselt werden, so daß von außen nicht alle Details (von Funktion oder Struktur) sichtbar sind.

Infrastruktur Alle für die Funktionsfähigkeit der Wirtschaft eines Landes notwendigen Verhältnisse, Einrichtungen und Anlagen, z. B. Arbeitskräfte, Straßen, Kanalisation.[WB99]

Informatiker Mit der Informatik beschäftigter Wissenschaftler, Techniker [WB99].

Informatikerin Mit der Informatik beschäftigte Wissenschaftlerin, Technikerin [WB99].

Interface builder siehe Anordnungswerkzeug

Komponenten Lauffähige Anwendungen (meist als Binärcode vorliegend), die zu größeren Anwendungen „zusammengesteckt" werden können. Die Schnittstellen sind fest definiert und es gibt Mechanismen, die das Verbinden der Schnittstellen übernehmen. Im Unterschied zu Komponenten wird bei der Verwendung von *Anwendungsrahmen* der Code von Klassenbeschreibungen und möglichen Interaktionen verwendet und letztendlich die Anwendung unter Verwendung der Klassen des Anwendungsrahmens implementiert.

Modell „Eine semantisch vollständige Abstraktion eines Systems, also eine vollständige und selbstbeschreibende Vereinfachung der Realität, die man wählt, um das System besser beschreiben zu können" ([BRJ97], S. 93).

Multiview Environment Entspricht den integrierten Entwicklungsumgebungen. Es gibt mehrere Ansichten auf ein Repository, das den Entwicklungsstand speichert.

Oberflächenobjekte Sind die Objekte, aus denen die Benutzungsoberfläche besteht.

Prinzip Grundsatz, Regel, Richtschnur [WB99]; Grundsatz, den jemand seinem Handeln und Verhalten zugrundelegt, allgemeingültige Regel, nach der etwas abläuft [Mül85].

Programmbibliothek Eine Sammlung von Programmstücken, die z. B. durch Aufruf oder Erben in neue Programme eingebunden werden können

Spezifikation Detaillierte Beschreibung der Teile eines Ganzen und ihrer Eigenschaften[Sch97b]. Auch ein Programm ist danach eine Spezifikation, oft wird der Begriff aber nur für abstraktere Beschreibungen benutzt oder für Beschreibungen, die nur Teilaspekte eines Programms spezifizieren.

Spezifikationsmethode Der Begriff Spezifikationsmethode wird in dieser Arbeit verwendet, um eine Spezifikationssprache oder -technik zusammen mit der Vorgehensweise zur Anwendung der Sprache oder Technik zu verwenden. Falls vorhanden, gehört auch die Realisierung durch ein Werkzeug dazu.

Spezifikationssprache Mittel zur sprachlichen und/oder graphischen Darstellung von Programmen und Systemen und ihrer Eigenschaften und Verhaltensweisen. Im Gegensatz zu einer Programmiersprache werden in einer Spezifikationssprache nicht die genauen algorithmischen Abläufe und die konkreten Datenstrukturen, sondern nur die gewünschten Eigenschaften des herzustellenden Produkts beschrieben. Spezifikationssprachen werden in der Entwurfsphase, also noch vor dem Einsatz von Programmiersprachen, benutzt. [Wis93]. Moderne Ansätze gehen davon aus, daß ein System auf einem, dem Denken der Benutzer/innen sehr nahen Niveau, beschreiben werden sollte (nach [Sch97b]).

Spezifikationstechnik Notation zur Spezifikation von Systemen [Sch97b].

Stöberer (engl. *browser*) Ein Werkzeug, daß das Stöbern in einer großen Datenmenge erlaubt, z. B. für Webseiten oder Programmtext.

System Sammlung von Untersystemen (sog. Subsystemen), die so organisiert sind, daß sie einen Zweck erfüllen. Systeme werden durch Modelle, evtl. Modelle aus verschiedenen Sichten, beschrieben. Untersysteme sind wiederum Sammlungen von Elementen, die zusammen ein bestimmtes Verhalten realisieren [BRJ97] S. 93.

Verbindungsobjekte sind die Objekte, die Oberfläche und Anwendung verbinden. Synonym: Adaptoren (Smalltalk- und Komponententechnologie-Jargon).

Widget Ein Oberflächenelement. Der Name ist eine Zusammenziehung von Window und Gadget (Ding), bezieht sich also auf alle Objekte, die auf einer Benutzungsoberfläche zu sehen sind.

Anhang A

Werkzeuge zur Entwicklung von Benutzungsschnittstellen

Das Entwerfen von Benutzungsschnittstellen wird durch verschiedenartige Werkzeuge unterstützt. Diese Werkzeuge sind in unterschiedlichen Bereichen entstanden: aus der Informatik (z. B. Programmierung, Software Engineering), aus den Design-Disziplinen (Benutzungsschnittstellen-Design, Web-Design) und aus den verschiedensten Anwendungsgebieten. Brad Myers hat auf einer Web-Seite (siehe [Mye00b]) über 150 Werkzeuge aufgelistet, und das sind nur die verfügbaren; viele der Forschungsansätze sind trotz guter Ideen nie für einen Vertrieb fertiggestellt worden. Aufgrund der Vielzahl der Werkzeuge soll hier nur eine kurze Beschreibung verschiedener Klassen von Werkzeugen, zusammen mit ausgewählten Beispielen, gegeben werden, die jeweils typische Vertreter dieser Klasse repräsentieren. Dabei handelt es sich nicht immer um die kommerziell erfolgreichsten oder, aus wissenschaftlicher Sicht gesehen, besten Werkzeuge ihrer Art, sondern um, in dieser Arbeit untersuchte, typische Exemplare. Eine in irgendeiner Hinsicht vollständige Auflistung würde den Rahmen der Arbeit sprengen.
Die Klassifikationen basiert auf den Arbeiten von [Pöp96, Mye95, Mye00b].
Ich unterscheide die Werkzeuge nach

1. Bibliotheken, die vordefinierte Elemente für Benutzungsschnittstellen enthalten,
2. Werkzeugen, deren Hauptaufgabe der Entwurf von Benutzungsschnittstellen oder die Entwicklung von Elementen von Benutzungsschnittstellen ist,
3. Anwendungswerkzeugen, die hauptsächlich zur Entwicklung von Anwendungen bestimmter Bereiche dienen, insbesondere aus technischen Gebieten, wie es dem Schwerpunkt dieser Arbeit entspricht und
4. integrierenden Werkzeugen, die eine Kombination der anderen Werkzeuge beinhalten.

A.1 Bibliotheken

Werkzeugkästen

Eine Bibliothek von Widgets wird oft auch Toolkit (Werkzeugkasten) genannt, genauso wie die Schicht der Benutzungsschnittstelle, die aus diesen besteht. Jede Entwicklungsumgebung,

mit der Benutzungsschnittstellen erstellt werden können, enthält einen solchen Baukasten. Oft müssen neue Widgets programmiert und der Bibliothek hinzugefügt werden. Es gibt allgemeine Werkzeugkästen, z. B. für Fenster, Rollbalken etc. und anwendungsspezifische Werkzeugkästen, z. B. für Simulation. Beispiele[1] für kommerzielle Werkzeugkästen sind das **X Toolkit** [Jos94] und **Motif [Lee00]**. Die Werkzeugsammlung von **Amulet**, das später unter dem Namen **Garnet** reimplementiert wurde [MFM+96], basiert auf solchen Werkzeugkästen. Es gibt auch anwendungsspezifische Werkzeugkästen; Beispiele dafür werden in Abschnitt A.3 aufgelistet.

Anwendungsrahmen

Anwendungsrahmen sind Klassenbibliotheken, in denen neben den einzelnen Elementen auch Beziehungen für Anwendungen implementiert sind (siehe auch Abschnitt 6.5).

Mit Hilfe der durch den Anwendungsrahmen **HotDoc** [Buc98] definierten Parts, können aktive Dokumente, entsprechend Webseiten, zusammengestellt werden. Das **Application Framework** von VisualWorks stellt alle wichtigen Klassen und die Mechanismen und Infrastruktur zur Entwicklung von traditionellen, zweidimensionalen Benutzungsschnittstellen zur Verfügung.

Komponentenrahmen

Komponentenrahmen dienen zum Einstecken von Komponenten, wie in Abschnitt 4.4.3 vorgestellt. Es gibt dabei Rahmen, die nur die Komponenten selbst und ihre Kommunikation enthalten, daher werden sie hier unter Bibliotheken aufgeführt.

Java Swing [WC99] ist ein Komponentenrahmen, mit dem 2D-Benutzungsschnittstellen entworfen werden können. Im weiteren Sinne ist Java Swing auch ein Anwendungsrahmen, da auch größere Komponenten, die eine Anwendung implementieren, enthalten sind (z. B. eine Tabellenkalkulation).

A.2 Benutzungsschnittstellen-Entwicklungssysteme

Anordnungswerkzeuge (interface builder)

Die Elemente, die in den Werkeugkästen definiert sind, müssen für die Präsentation auf der Benutzungsoberfläche angeordnet werden. Interface Builder erlauben die Anordnung auf intuitive Weise, indem sie die Widgets bereits graphisch repräsentieren (auf einer Palette) und die graphischen Eigenschaften und die Anordnung durch graphische Manipulation spezifizieren können.

Der **UIBuilder** von VisualWorks erlaubt das Anordnung vordefinierter, klassicher zweidimensionaler Widgets. **Gipsy** [AG00] ist ein Anordnungswerkzeug für vordefinierte Widgets verschiedener Anwendungsgebiete.

[1] Wie bereits erwähnt, werden hier nur wenige, typische Beispiele gegeben. Es sind viele Werkzeugkästen entwickelt worden. Eine Auflistung würde den Rahmen der Arbeit sprengen.

Visuelle Sprachen für Benutzungsschnittstellen

Insbesondere technische Systeme (siehe unten, Abschnitt A.3) bieten diagrammartige visuelle Sprachen für die Entwicklung von Anwendungen an, da sich die Signalflüsse und Blöcke einer Anwendung gut durch Datenfluß- und Blockdiagramme darstellen lassen. Oft werden auch die Elemente der Benutzungsschnittstelle als Blöcke dargestellt.

Visual Designer [Int00], **Simulink** [Sci00] und **LabView** [Nat00](siehe Abschnitt A.3) basieren auf Block- und Datenflußdiagrammen.

Visuelle Allzweck-Programmiersprachen wie **Prograph [SC95]** können auch für die Programmierung von Benutzungsschnittstellen verwendet werden.

Visuelle Komponentenrahmen

Einige Komponentenrahmen haben eine Visualisierung, mit der die Anordnung von Komponenten ähnlich den Interface Buildern möglich ist. Die Komponenten werden oft durch Piktogramme repräsentiert, insbesondere die nicht sichtbaren Komponenten, siehe 4.4.3(z. B. die Bibliothek der Programmiersprache Java[CW98]).

JavaStudio ist ein rein visuelles Werkzeug. Die Entwicklung findet statt mit einem Layout-Werkzeug, mit dem sichtbare Komponenten zusammengesetzt werden können und einem visuellen Datenflußwerkzeug, mit dem die Funktionalität aus der Anordnung und Verbindung von visuellen und nichtvisuellen Komponenten. Dadurch kann die ganze Anwendung programmiert werden. Die **BeanBox** ist ein einfaches Werkzeug zum Zusammensetzen einer Benutzungsschnittstell aus Komponenten [CWHT98].

Prototypensysteme

Dienen zur schnellen Erstellung der Präsentation, ohne daß die Funktionalität implementiert sein muß. Im weiteren Sinne sind auch mit Papier und Bleistift erstellte Skizzen Prototypen. Prototypensysteme sind alle Software-Systeme, die ein Bild oder eine Folge von Bildern liefern, die das Aussehen der Benutzungsschnittstelle beschreiben. Daher gehören Anordnungswerkzeuge, visuelle Komponentenrahmen, Skizzensysteme usw. zu den Prototypensystemen.

Im engeren Sinne zeigen Prototypensysteme auch das Oberflächenverhalten der Benutzungsschnittstelle. **VisualBasic** [Mic00b] ist ein Vertreter solcher Systeme, bei denen ein Anwendungswerkzeug mit der Generierung von Programmiertext-Schablonen einer einfachen Programmiersprache integriert ist.

Skizzensysteme

Im Gegensatz zu den Prototypensystemen, bei denen es nur auf den optischen Eindruck, d. h. die Präsentation ankommt, geht es bei den Skizzensystemen um die schnelle und einfache Eingabe von Skizzen von Benutzungsschnittstellen. Dazu wird z. B. Gestenerkennung verwandt.

SILK erlaubt das Skizzieren von Benutzungsoberflächen: Das System errät aus der Form der gemalten Oberflächenelemente wird abgeleitet, welche Widgets gemeint sind (z. B. kleine Kreise werden als Knöpfe interpretiert). Weiterhin können Folgen von solchen Skizzen zu einem Demonstrator zusammengesetzt werden. **Knight** [DHT00] ist ein Werkzeug für Objektorientierte Analyse, das UML-Elemente aus den Skizzen ableitet.

Programmieren durch Beispiele oder durch Vormachen

Systeme, die Programmieren durch Beispiele oder durch Vormachen anbieten, leiten aus (unvollständigen) Eingaben der Entwickler/in ab, welche Programmieraktion „gemeint" ist. D. h. die Entwickler/in kann unvollständige Eingaben machen. In der Regel sind solche Systeme mit Hilfe von Constraints realisiert.

Klassische Vertreter dieser Systeme sind **ThingLab** [Bor78] und **Peridot** [MMK], bei denen das jeweilige System die Anordnung und damit Teile des graphischen Verhalten von Benutzungsschnittstellen aus Beispielen ableitet, die die Entwickler/in dem System gibt. Es existiert eine Reimplementierung unter dem Namen **Marquise** [MMK93], die in die oben vorgestellte Werkzeugsammlung Garnet integriert ist.

Modellbasierte Werkzeuge

Der Begriff *modellbasiert* wird unterschiedlich interpretiert, allgemein drückt er aus, daß zuerst ein Modell der Benutzungsschnittstelle definiert wird und daraus eine lauffähige Version generiert wird. Das *Modell* kann auf verschiedenen Stufen des Entwurfs angesiedelt sein: Werkzeuge, die auf einem Aufgabenmodell basieren [SB99], werden ebenso modellbasiert genannt wie Werkzeuge, die aus einer deklarativen Beschreibung der Funktionsweise einer Benutzungsschnittstelle generiert werden.

Der **Teallach Model** Ansatz [GPGW99] basiert auf drei Modellen: dem Aufgabenmodell, dem Bereichsmodell und dem Präsentationsmodell. Modelle können durch eine baumartige Strukturen aller Objekte dargestellt werden. Das Werkzeug erleichtert die Verbindung von Anwendungsmodell und Aufgabenmodell, indem die Aufgabenmodellzustände mit den Operationen des Anwendungsmodells verknüpft werden können. Das Werkzeug **autoCAID** [Züh00] auch deshalb interessant, weil es im Bereich Maschinenbau als Anwendungsbereich entwickelt wurde. Es umfaßt verschiedene Werkzeuge, die deklarative, graphische und objektorienterte Spezifikationen in einem Repository zu einem Modell zusammenfassen. Die Werkzeuge müssen in einer vorgegebenen Reihenfolge, den sog. Gestaltungsschritten verwendet werden, um die Entwickler/in durch den Prozess zu führen. Auch Benutzungsschnittstellen-Verwaltungssysteme (siehe Abschnitt A.5), sind oft modellbasiert, z.B. **UIDE** [Suk93, FF94] oder **MOBI-D** [PS94]. Auch generierende Werkzeuge benutzen Modelle: Das Werkzeug **GENIUS** [JWZ93], das leider nicht mehr weiterentwickelt wird, benutzte Entity-Relationship-Modelle und Petrinetze als Modelle.

A.3 Werkzeuge aus den Anwendungsgebieten

Mathematische Bibliotheken und Systeme

Mathematische Bibliotheken und Systeme bieten oft auch Funktionen und Widgets zur Erstellung von Benutzungsschnittstellen an. Historisch ist dies durch die Verwendung mathematischer Visualisierungen bedingt, z. B. durch Funktionsgraphen. Heute sind auch Standard-Elemente wie Knöpfe und Eingabefelder als Widgets verfügbar.

Trotzdem sind diese Werkzeuge oft auf bestimmte Anwendungsbereiche spezialisiert oder ganz im Anwendungsfeld Mathematik geblieben.

Da sich Prozeßmodelle in technischen Anwendungen oft durch mathematische Funktionen und Gleichungen ausdrücken lassen, liegt es für die Anwender/innen solcher mathematischer Bibliotheken und Systeme häufig nahe, sie für die Entwicklung der Benutzungsschnittstellen zu verwenden.

Mathematika ist aus einer Bibliothek für mathematische Konstrukte entstanden. Formeln können graphisch interpretiert werden und bilden so, zusammen mit Interaktionsobjekten eine Benutzungsschnittstelle [Wol00]. **MATLAB** [Sci00] hat ebenfalls eine mathematische Bibliothek als Grundlage. Zusammen mit einer C-ähnlichen Programmiersprache entsteht eine Programmierumgebung mit Visualisierungsmöglichkeiten.

Technische Bibliotheken und Systeme

Es gibt allgemeine Werkzeugkästen, z. B. für Fenster, Rollbalken etc. und anwendungsspezifische Werkzeugkästen, z. B. für Simulation (Erweiterungen von MATLAB [Sci00]) oder Prozeßleitsysteme ([in-00a]).

Simulink [Sci00] baut auf MATLAB auf, es benutzt dessen mathematische Bibliothek, erlaubt aber auch die Erstellung von Blockschaltbildern, mit Hilfe eines weiteren Werkzeugs auch zusätzlich die Programmierung mit Hilfe von Zustands-Übergangsdiagrammen. **LabView** [Nat00] enthält vordefinierte Widgets und Visualisierungen für die Meßtechnik.

Autorensysteme

Autorensysteme sind eigentlich für die Erstellung von Dokumenten gedacht, werden jedoch oft für die Entwicklung von Prototypen verwendet, insbesondere im HCI-Bereich.

Toolbook [cli00] ist ein Werkzeug, mit dem Lernumgebungen leicht realisiert werden können, indem interaktive Benutzungsschnittstellen-Elemente in Dokumente plaziert werden können.

A.4 CASE-Werkzeuge

Mit Werkzeugen für das computerunterstützte Entwerfen von Software (*Computer Aided Software Engineering*, CASE) können Benutzungsschnittstellen auf zwei Arten spezifiziert werden:

Modellierungswerkzeuge

erlauben das Entwerfen von Klassen und ihren Beziehungen und damit auch den Aufbau einer Benutzungsschnittstelle als Klassenhierarchie. Diese Form wird allerdings selten genutzt, normalerweise wird die Benutzungsschnittstelle in einer späteren Phase mit einem Benutzungsschnittstellen-Werkzeug erstellt und dann an das von dem Modellierungswerkzeug erzeugte Programm angebunden. Das in dieser Arbeit erstellte Werkzeug COMBO erlaubt allerdings beide Vorgänge in einer Phase.

RationalRose [Rat00] unterstützt die UML und ist ein häufig verwendetes Werkzeug. Neben Werkzeugen, die sich auf Methode festlegen, gibt es auch Meta-Werkzeuge, die erlauben, eine Methodik und die dazugehörigen Notationen selbst zu definieren.

Generierende CASE-Werkzeuge

Diese Werkzeuge entwerfen die Objektstruktur der Anwendung, wie die Modellierungswerkzeuge, und generieren nach fest vorgegebenen Regeln aus einzelnen Datenstrukturen eine Benutzungsschnittstelle, z. B. für jede Klasse der Datenstruktur ein Fenster, in dem die Attribute von Ausprägungen dieser Klasse gesetzt werden können.

Janus [Bal93] ist ein Werkzeug, das Benutzungsschnittstellen aus einem OOA-Modell generiert. Die OOA-Modelle werden mit einem Modellierungswerkzeug entworfen. Der Schwerpunkt der Modellierung ist die Modellierung der Anwendungsdaten, das Werkzeug generiert Fenster aus jeder Klasse, die im Ergebnis formularartig sind.

A.5 Integrierende Werkzeuge

Programmierumgebungen

Programmierumgebungen basieren zunächst auf Editoren für eine bestimmte Programmiersprache. Oft werden aber auch Navigationshilfen (z. B. Klassenstöberer) oder Zusatzprogramme für visuelle Programmierung angeboten. Zu dieser Klasse von Werkzeugen gehören z.B. **VisualWorks** [Par95] für Smalltalk, **VisualAge** [**IBM00**] für Smalltalk und Java, die in Abschnitt 4.5 vorgestellt wurden.

Benutzungsschnittstellen-Entwicklungswerkzeuge (*user interface development systems*)

Unter dem Begriff *Benutzungsschnittstellen-Entwicklungswerkzeuge (user interface development systems*, UIDS) werden alle Werkzeuge zusammengefaßt, die den Entwurf von Benutzungsschnittstellen durch spezielle Sprachen, graphische oder sonstige Methoden unterstützen. Dazu gehören Interface-Builder, Werkzeuge zur Erstellung von Prototypen, Entwurfsumgebungen für das Programmieren in einer bestimmten Programmiersprache und Werkzeuge, die auf formalen oder ereignisbasierten Sprachen aufbauen.

Benutzungsschnittstellen-Verwaltungssysteme (*user interface management systems*)

Benutzungsschnittstellen-Verwaltungssysteme (*user interface management systems,* UIMS) sind UIDS's, die zusätzlich eine Laufzeitverwaltung beinhalten (entsprechen dem Begriff Datenbankmanagementsystem).

HUMANOID generiert Benutzungsschnittstellen aus einer deklarativen Beschreibung [SLN93]. Es baut auf dem oben beschriebenen Garnet-Toolkit auf. Eines der ersten Benutzungsschnittstellen-Verwaltungssysteme war **UIDE** (User Interface Development Environment). UIDE folgt dem modellbasierten Ansatz, ist aber auch ein UIMS. Benutzungsschnittstellen werden in UIDE textuell in der Sprache IDL (Interactive Description Language) spezifiziert [Suk93][dBFM92]. **TeleUse** [in-00b] für Benutzungsschnittstellen unter Motif und Windows wird kommerziell vertrieben. Es integriert auch Anordnungsblawerkzeuge und ander Hilfwerkzeuge, z.B. für die Anbindung an Datenbanken.

Anhang B

Fragebogen zum Thema: Entwicklungsumgebungen für Smalltalk und Java

Auf den folgenden Seiten wird der Fragebogen abgedruckt, der Grundlage für die Umfrage in Abschnitt 4.5.2 war.

Leser/innen, die Interesse daran haben, ihre Erfahrungen mit Entwicklungsumgebungen weiterzugeben, sind herzlich eingeladen, den Fragebogen auszufüllen.

Am einfachsten geht dies im WWW, unter der Adresse

http://www.pu.informatik.tu-darmstadt.de/siemon/fragebogen.html.

english version

"Visuelle" Entwicklungsumgebungen für Smalltalk und Java

Vielen Dank, daß Sie an der Umfrage zu diesem Thema teilnehmen. In diesem Fragebogen werden die visuellen Entwicklungsumgebungen hauptsächlich für Smalltalk und Java unter dem Schwerpunkt der Programmierung von Benutzungsoberflächen untersucht. Falls Sie eine Entwicklungsumgebung für eine andere Sprache benutzen, können Sie den Fragebogen zu Vergleichszwecken gerne auch ausfüllen.

Das Ausfüllen des Fragebogens dauert zwischen 10 Minuten und 1 Stunde, je nachdem, wie ausführlich Sie die Fragen beantworten.

Der Fragebogen besteht aus drei Teilen:

Teil I: Allgemeine Fragen zur Einordnung

1. Enwicklungsumgebung
2. Erfahrung
3. Benutzungsschnittstellen

Teil II: Fragen zum visuellen Anteil der Entwicklungsumgebung, Vor- und Nachteile.

4. Welche Möglichkeiten der visuellen Programmierung bietet die Entwicklungsumgebung
5. Benutzung
6. Interaktion mit dem/der Benutzer/in
7. *Kognitive Dimensionen*
8. Design Patterns
9.Komponenten

Teil III: Weitere Information

Auswertung des Fragebogens
Fallstudie

Teil I: Allgemeine Fragen zur Einordnung

1. **Benutzen Sie eine Entwicklungsumgebung für die Programmierung von Smalltalk oder Java?**
   Falls ja, welche Entwicklungsumgebung benutzen Sie? (wählen Sie aus oder tippen Sie den Namen ein) für
   VisualWorks Java
   Smalltalk

   (falls Ihre Entwicklungsumgebung nicht in der Liste enthalten ist, oder Sie zusätzliche Werkzeuge wie z.B. ein UML-Werkzeug verwenden, geben Sie hier
   den Name der Entwicklungsumgebung ein:
)

   Falls nein, warum nicht?

---

2. **Wieviel Erfahrung haben sie mit der Entwicklungsumgebung?**
   Ich habe bereits
   mehrere Aufgaben programmiert

---

3. **Haben Sie mit der Entwicklungsumgebung schon mal eine Benutzungsschnittstelle programmiert?**
   (falls ja, bitte ankreuzen)
   Bitte beschreiben Sie die Anwendung, für die Sie die Benutzungsschnittstelle programmiert haben, kurz:

Teil II: Fragen zum visuellen Anteil der Entwicklungsumgebung, Vor- und Nachteile.

4. **Welche Möglichkeiten der "visuellen Programmierung" für Benutzungsoberflächen bietet die Entwicklungsumgebung?**

Visuelle Unterstützung	**habe ich schon benutzt** **(bitte anklicken)**	**ist hilfreich (1= sehr ... 6 = gar nicht)**
Interface-Builder		
Klassenhierarchie wird graphisch angezeigt		
Es gibt eine Komponentenpalette auch für Datenobjekte (die nicht auf der Benutzungsoberfläche sind)		
Abhängigkeiten zwischen Komponenten können durch Linien spezifiziert werden		
es gibt eine visuelle Programmiersprache		

5. **Wieviel der oben von ihnen beschriebenen Benutzungsschnittstelle haben Sie unter Benutzung der visuellen Möglichkeiten programmiert?**
   (Sie können eine Prozentzahl angeben oder etwas schreiben)

6. Zur Verarbeitung von Eingaben der Benutzerin/des Benutzers bieten die meisten Framworks spezielle Klassen und/oder Mechanismen an (manchmal auch Event-Modell genannt).
   Wie unterstützt Ihre Entwicklungsumgebung die Programmierung der Verarbeitung von Eingaben?
   a. Benutzen Sie die vom Framwork vorgegebenen Klassen? (Falls ja, bitte ankreuzen)
      Bitte beschreiben Sie das Event-Modell/die vorgegebenen Klassen kurz:

   b. **Bietet die Entwicklungsumgebung besondere Methoden, Dialoge und Eingaben zu spezifizieren (z.B. Skripte, Dialogbeschreibungssprachen)?** Wenn ja, welche?

   c. **Kann die Verarbeitung von Benutzer/inneneingaben visuell spezifiziert werden? (z.B. durch visuelle Dialogbeschreibungssprachen, visuelle Scriptsprachen, Diagramme)**

7. Falls Ihre Entwicklungsumgebung visuelle Unterstützung in Form eines Interface-Builders, einer graphischen Darstellung der Klassenhierarchie, einer Komponentenpalette, einer visuellen Programmiersprache oder der Programmierung durch graphische Interaktion anbietet, beantworten Sie auch bitte die folgenden Fragen [1]. Dabei können Sie entweder etwas schreiben oder eine Bewertung angeben zwischen 0 und 10 angeben.
   a. **Finden Sie, daß die visuellen Möglichkeiten mit der Programmiersprache (Java oder Smalltalk) gut verträglich oder mußten Sie sozusagen eine neue Sprache lernen?**
      (10 = gut verträglich, 0 = ganz neue Sprache lernen)
      `Bewertung:  Kommentar`

   b. Angenommen, Sie würden ein Schulung für Programmieranfänger/innen, die aber Experten eines Anwendungsgebiets sind, zur Einführung in die Entwicklungsumgebung halten.
      1. Würde es genügen, den Schüler/innen den visuellen Teil zu erklären?
         (10 = ja, um Anwendungen zu schreiben, braucht man nur den visuellen Teil,
         0 = nein, Anfänger/innen würden mit den visuellen Möglichkeiten die Prinzipien überhaupt nicht verstehen)
         `Bewertung:  Kommentar`
      2. Ich würde folgende Teile einer Anwendung mit Hilfe der visuellen Anteile erklären (bitte anklicken):

Aussehen der Benutzungsoberfläche	
Verhalten der Benutzungsoberfläche	
Reaktion auf Ereignisse	
Verhalten des restlichen Programms	
Verbindung von Oberfläche und dahinterliegender Funktionalität	
Entwurf von Klassen	
Methodenprogrammierung	

   c. **Haben Sie den Eindruck, daß die Interaktion mit den visuellen Möglichkeiten intuitiv ist** (d.h., Sie beispielsweise leicht aus bereits Bekanntem ableiten können, wie eine neue Aufgabe zu lösen ist)?
      (10 = intuitiv -- 0 = schwer zu verstehen und merken)

      `Bewertung:  Kommentar`

   d. Bei der rein textuellen Programmierung gibt es Fehler, die immer wieder passieren, z.B. das Vergessen eines Semikolons. **Treten bei der Programmierung mit den visuellen Möglichkeiten Ihrer Entwicklungsumgebung solche Fehler ebenfalls auf**?
      (10 = nein, nie , 0 = ja, ständig)
      `Bewertung:  Beispiele: Kommentar:`

   e. Viele Elemente von Benutzungoberflächen sind abhängig von anderen Elementen oder von anderen Datenstrukturen (z.B. kann ein Textfeld abhängig sein von einer Liste, indem im Textfeld immer das selektierte Listenelement zu sehen sein soll). **Sind solche Abhängigkeiten (graphisch) zu sehen oder existieren Sie irgendwo versteckt im Source-Code?**

(10 = alle Abhängigkeiten sichtbar, 0 = alle Abhängigkeiten unsichtbar)
`Bewertung: Kommentar:`

f. **Ist die visuelle Entwicklungsumgebung änderungsfreundlich?** Oder müssen Sie wie beim Schach zehn Züge im voraus überlegen, damit sie hinter z.B. keine Namenskonflikte bei der Benennung von Widgets bekommen oder ein Layout haben, das eher an Spaghetti als an ein Programm erinnert?
(10 = änderungsfreundlich, 0 = nach dem Aufschreiben ist keine Änderung mehr möglich)
`Bewertung: Kommentar:`

Wie einfach ist es, den gesamten Entwurf zu ändern?
(10 = der gesamte Entwurf kann an jeder Stelle jederzeit leicht abgeändert werden,
5 = z.B. man muß viel mit der Maus klicken, abr es geht ,
0 = wenn eine Entwurfsphase abgeschlossen ist, ist nichts mehr zu ändern)
`Bewertung: Kommentar:`

Wie hoch ist der Aufwand, um eine einzelne Änderung einzufügen?
(10 = gering, z.B. nur ein Mausklick auf das zu ändernde Element, 0 = hoch)
`Bewertung: Kommentar:`

g. **Können Sie ihre Benutzungsoberfläche während der Entwicklung ausprobieren?**
(10 = Benutzungsoberfläche kann zu jedem Zeitpunkt ausprobiert werden, 0 = Benutzungsoberfläche muß erst vollständig programmiert werden)

`Bewertung: Kommentar:`

h. **Wie gut ist Ihrer Meinung nach die Übersicht über die ganze Anwendung**? Haben Sie zu jedem Zeitpunkt die Übersicht, in welchen Kontext das Element, das Sie gerade bearbeiten, hineingehört? Oder sind Sie ständig am Hin- und Hersuchen zwischen graphischer Darstellung und Source-Code?
(10 = gute Übersicht, 0 = keine Übersicht)
`Bewertung: Kommentar:`

i. Bei der visuellen Programmierung spricht man von *sekundärer Notation*, wenn Eigenschaften wie Farbe und Größe eine Unterschied in der Bedeutung ausmachen. Z.B ist eine blaue Verbindungslinie an, daß es sich um eine Attributbeziehug handelt und eine grüne, daß es sich um einen Methodenaufruf aufgrund eines Events handelt. **Benutzt Ihre Entwicklungsgebung solche sekundäre Merkmale?**
(10 = viele sekundäre Merkmale, 0 = keine sekundären Merkmale)
`Bewertung: Kommentar:`

Halten Sie diese sekundäre Notation für hilfreich?
(10 = sehr hilfreich, 0 = nicht hilfreich)
`Bewertung: Kommentar:`

---

8. **Unterstützt die Entwicklungsumgebung Design Patterns?** (Falls ja, bitte ankreuzen)
Bitte beschreiben Sie die Art der Unterstützung (z.B. Generierung von Schablonen):

---

9. **Unterstützt die Entwicklungsumgebung die Entwicklung von Komponenten?** (Falls ja, bitte ankreuzen)
Bitte beschreiben Sie die Art der Unterstützung (z.B. Einbinden von eigenen Komponenten in eine Komponentenpalette):

Das war's!

Teil III: Weitere Information

Vielen Dank für das Ausfüllen des Fragebogens! Bevor Sie die Antworten abschicken, hier noch ein paar Informationen:

Fallstudie

Ich werden Anfang nächsten Jahres eine Fallstudie zu diesem Thema durchführen. Dabei geht es darum, mit dem von Ihnen benutzten Werkzeug unter Beobachtung Teile einer Benutzungsoberfläche zu programmieren.
Dafür suche ich noch Testpersonen.

Haben Sie Interesse daran, an der Fallstudie teilzunehmen?
Nein

Auswertung des Fragebogens

Ich werde den Fragebogen bis Ende Januar auswerten.Wenn Sie an dem Ergebnis interessiert sind oder an der Fallstudie teilnehmen möchten, hinterlassen Sie bitte Ihren Namen und Ihre Adresse. Diese Daten und die Antworten des Fragebogens werden nur für diesen Zweck und nur intern verwendet.
`Name: Telefonnummer: email: Fax: Postadresse:`

Weitere Informationen über meine Arbeit finden Sie unter
www.pu.informatik.tu-darmstadt/~siemon/CurrentWork/current.html

Haben Sie Anmerkungen oder Fragen zu diesem Fragebogen?

Submit

Elke Siemon, Technische Universität Darmstadt, Fachgebiet Programmiersprachen und Übersetzer, Alexanderstr. 10, 64287 Darmstadt, Tel: 06151-163710,

Abbildungsverzeichnis

Tabellenverzeichnis

Index

()

Literaturverzeichnis

[ABBK93] Ben Abbott, Tedd Babty, Csaba Biegl und Gabor Karsai: *Model-Based Software Synthesis*. IEEE Software, Seite 43–51, Mai 1993.

[AG00] Patzschke + Rasp Software AG: *GIPSY 4.1: Kürzere Entwicklungszeiten, umfangreiche Symbolbibliothek und unbegrenzte Anzahl von Objekten*. Run Times Flash, 2000.

[Ale79] Christopher Alexander: *The timeless way of building*. Oxford University Press, 20 Auflage, 1979.

[AIS77] Christopher Alexander, Sara Ishikawa und Murray Silverstein: *A pattern language*. Oxford University Press, 30 Auflage, 1977.

[ABW98] Sherman R. Alpert, Kyle Brown und Bobby Woolf: *The design patterns Smalltalk companion*. Addison Wesley Longman, 1998.

[Bal93] Helmut Balzert: *Der JANUS-Dialogexperte: Vom Fachkonzept zur Dialogstruktur*. In: *Softwaretechnik '93*, Seite 62–72, 1993.

[BP90] Remi Bastide und Philippe Palanque: *Petri Net Objects for the Design, Validation and Prototyping of User-Driven Interfaces*. In: *Human-Computer Interaction*, Seite 625–631. Elsevier Science, 1990.

[BP95] Remi Bastide und Philippe Palanque: *A Petri Net Based Environment for the Design of Event-Driven Interfaces*. In: Giorgio De Michelis (Herausgeber): *Theory and Applications of Petri Nets*, Springer Informatik, Seite 278–297. Springer Verlag, 1995.

[Bau90] Bernd Baumgarten: *Petri-Netze: Grundlagen und Anwendungen*. BI Wissenschaftsverlag, Mannheim, Wien, Zürich, 1990.

[Bec00] Kent Beck: *Extreme programming explained*. Addison Wesley, 2000.

[BM93] Ulrich Becker und Daniel Moldt: *Objektorientierte Konzepte für gefärbte Petri-Netze*. Advances and Theory of Petri Nets, Seite 140–151, 1993.

[BL93] Brigham Bell und Clayton Lewis: *ChemTrains: A Language for Creating Behaving Pictures*. In: *Proceedings of the IEEE Symposium on Visual Languages*, Seite 188–195, August 1993.

[Bis00] Michael-Georg Bistekos: *Webseite der Firma Software-Ergonomie.com*. http://www.software-ergonomie.com, 2000. referenziert am 25.5.2000.

[Bol80] Richard A. Bolt: *Put that there: Voice and Gesture at the Graphics Interface.* ACM Computer Graphics, 14(3) Seite 262–270, August 1980.

[BRJ97] Grady Booch, James Rumbaugh und Ivar Jacobson: *The Unified Modeling Language User Guide.* Addison Wesley, 1997.

[Bor00a] Jan Borchers: *CHI meets PLoP: An interaction patterns workshop.* ACM SIGCHI Bulletin, 32(1) Seite 8–12, Januar 2000.

[Bor00b] Jan Borchers: *A Pattern Approach to Interaction Design.* Dissertation, Technische Universität Darmstadt, Mai 2000.

[BFG+99] Jan Borchers, Jane Finlay, Richard Griffiths, Elke Siemon und Lyn Pemberton: *Usability Pattern Language: Creating a Community.* In Vorbereitung, Stand von Mai 2000, 1999.

[Bor78] Alan Borning: *ThingLab – A Constraint-Oriented Simulation Laboratory.* Technischer Bericht, Xerox Palo Alto Research Center, 1978. verfügbar unter http://www.2share.com/thinglab/ThingLab, referenziert am 15.11.2000.

[Bor86] Alan Borning: *Defining Constraints Graphically.* In: *CHI'86 Proceedings*, Seite 137–143, April 1986.

[Bos98] Jan Bosch: *Design Patterns as Language Constructs.* Journal of Object Oriented Programming, 11(2) Seite 18–32, 1998.
verfügbar unter http://www.pt.hk-r.se/~bosch, referenziert am 27.2.2000.

[Bra95] John Michael Brant: *HOTDRAW.* Diplomarbeit, University of Illinois at Urbana-Champain, 1995.

[BK93] M. Brielmann und B. Kleinjohann: *A Formal Model for Coupling Computer-Based Systems and Physical Systems.* In: *Proceedings of the European Design and Automation Conference with EURO-VHDL '93*, Seite 158–163. IEEE Computer Society Press, September 1993.

[Buc98] Jürgen Buchner: *HotDoc. Ein flexibles System für den kooperativen Aufbau zusammengesetzter Dokumentstrukturen.* Dissertation, Technische Universität Darmstadt, 1998.

[BFVY96] F. J. Budinsky, M. A. Finnie, J. M. Vlissides und P. S. Yu: *Automatic code generation from design patterns.* Systems Journal, IBM Object technology, 35(2), 1996. verfügbar unter
http://www.research.ibm.com/journal/sj/budin/budinsky.html,
referenziert am 23.7.2000.

[BCA00] Margaret Burnett, Nanyu Cao und John Atwood: *Time in Grid-Oriented VPLs: Just Another Dimension?* In: *2000 IEEE International Symposium on Visual Languages*, Seite 137–144, September 2000.

[BB94] Margaret M. Burnett und Marla J. Baker: *A Classification System For Visual Programming Languages.* Technischer Bericht, Oregon State University, 1994.

[BGL96] Margaret M. Burnett, Adele Goldberg und Ted Lewis (Herausgeber): *Visual Object-Oriented Programming: Concepts and Environments*. Manning Greenwich, 1996.

[BMR+96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad und Michael Stal: *Pattern Oriented Software Architecture, A System of Patterns*. John Wiley & Sons, 1996.

[CL00] Cooperative Computing & Communication Laboratory C-Lab: *C-Lab Pathfinder*. http://www.c-lab.de/~pathfinder, 2000. referenziert am 10.1.2000.

[CW98] Mary Campione und Kathy Walrath: *The Java (TM) Tutorial Second Edition: Object-Oriented Programming for the Internet*. The Java Series. Addison-Wesley, 1998.

[CWHT98] Mary Campione, Kathy Walrath, Alison Huml und Tutorial Team: *The Java (TM) Tutorial Continued: The Rest of the JDK (TM)*. The Java Series. Addison-Wesley, 1998.

[Cin00] Cincom Systems Inc.: *Visual Smalltalk Enterprise*. http://www.cincom.com/vse/prtwrap.html, Februar 2000.

[cli00] click2learn.com Inc.: *ToolBook*. http://home.click2learn.com/company/index.html, 2000. referenziert am 16.9.2000.

[CN94] Peter Coad und Jill Nicola: *Objekt-orientierte Programmierung*. Prentice-Hall, 1994.

[CNM95] Peter Coad, David North und Mark Mayfield: *Object Models*. Yourdon Press Computing Series. Prentice Hall Inc, 1995.

[CY91] Peter Coad und Edward Yourdon: *Object Oriented Analysis*. Yourdon Press, 1991.

[Col94] Dave Collins: *Designing Object-Oriented User Interfaces*. Benjamin Cummings, 1994.

[Con94] Concept asa GmbH: *THE WIDGET FACTORY*. Werbebroschüre, 1994.

[CS95] James Coplien und Douglas Schmidt (Herausgeber): *Pattern Languages of Program Design*. Addison Wesley, 1995.

[Cor00] Corel Corporation: *CorelDraw 10.0 Graphics Suite*. http://www.corel.com, 2000. referenziert am 21.11.2000.

[CNS94] J. Coutaz, L. Nigay und D. Salber: *A Reference Model for the Software Design of Interactive Systems*. Amodeus Project Document, Juni 1994.

[CBY00] Philip T. Cox, Omid Banyasad und June Young: *Constructing Robot Control Programs by Demonstration*. persönliches Gespräch, September 2000.

[CS96] Philip T. Cox und Trevor J. Smedley: *A Visual Language for the Design of Structured Graphical Objects*. In: *Proceedings of the IEEE Symposium on Visual Languages*, Seite 296–303, 1996.

[Cun00] Ward Cunnningham: *History Of Patterns*.
http://c2.com/cgi-bin/wiki?HistoryOfPatterns, 2000.
referenziert am 25.11.2000.

[Cyp93] Allan Cypher (Herausgeber): *Watch What I Do: Programming by Demonstration*. MIT Press, Cambridge, London, 1993.

[DHT00] Christian Heide Damm, Klaus Marius Hansen und Michael Thomsen: *Tool Support for Cooperative Object-Oriented Design: Gesture Based Modeling on an Electronic Whiteboard*. In: Thea Turner, Gerd Szwillus, Mary Czerwinski und Fabio Paterno (Herausgeber): *CHI 2000 Conference Proceedings*, Seite 518–525, April 2000.

[dBFM92] Dennis J.M.J. de Baar, James D. Foley und Kevin E. Mullet: *COUPLING APPLICATION DESIGN AND USER INTERFACE DESIGN*. In: *CHI'92 Proceedings*, Seite 259–266, Mai 1992.

[dW99] Kees de Weerd: *Bauanleitung: Die Adler* .
http://utopia.knoware.nl/users/cdeweerd/e24.html, 1999.
referenziert am 20.8.1999.

[DH94] Iris Dilli und Hans-Jürgen Hoffmann: *Diades-II: A Multi-Agent User Interface Design Approach with an Integrated Assessment component*. In: *Unterlagen der Special Interest Group „Tools for working with guidelines", CHI '94 Conference on Human Factors in Computing Systems*, April 1994.

[DVH93] Iris Dilli, Gregor Vogt und Hans-Jürgen Hoffmann: *Diades-II: Ein objektorientiertes Entwurfswerkzeug für interaktive Benutzungsschnittstellen auf der Grundlage koagierender Agenten*. Technischer Bericht PU1R6/93, Technische Hochschule Darmstadt, Juni 1993.

[Do95] E. Y.-L. Do: *What's in a diagram that a computer should understand*. In: M. Tan und R. Teh (Herausgeber): *CAAD Futures '95: The Global Design Studio, Sixth International Conference on Computer Aided Architectural Design Futures*, Seite 191–199, 1995.

[DW98] Desmond Francis D'Souza und Alan Cameron Wills: *Objects, Components, and Frameworks with UML*. ADDISON-WESLEY, 1998.

[Ege92] Raimund K. Ege: *Designing Maintainable, Reusable Interface*. IEEE Software, 9(6) Seite 24–32, November 1992.

[Eis00] Ralf Eissler: *Vorgehensweise zur Gestaltung von Bediensystemen*. In: Detlef Zühlke (Herausgeber): *Teilnehmerunterlagen zum Seminar Mensch-Maschine-Interaktion in komplexen technischen Systemen, Kaiserslautern*, 2000.

[ES95] T. Elwert und E. Schlungbaum. In: P. Palanque und R. Bastide (Herausgeber): *Design, Specification and Verification of Interactive Systems*, Seite 193–208, 1995.

[Fan99] GE Fanuc: *Klare Sachen.* IEE - Automatisieren, 44(S1) Seite 87–88, 1999.

[Fin00] Sally Fincher: *The Pattern Gallery.* http://www.cs.ukc.ac.uk/people/staff/saf/patterns/gallery.html, 2000. referenziert am 12.9.2000.

[fis96] fischerwerke Artur Fischer GmbH & Co KG: *LLWin - Lucky Logic für Windows*, Handbuch fischertechnik Auflage, 1996.

[FWS93] Vikki Fix, Susan Wiedenbeck und Jean Schotz: *Mental Representations of Programs by Novices and Experts.* In: *INTERCHI'93 Conference Proceedings (Amsterdam, Niederlande)*, Seite 74–79, 1993.

[FGKK88] James Foley, Christina Gibbs, Won Chul Kim und Srdjan Kovacevic: *A Knowledge-Based User Interface Management System.* In: *Proceeding der CHI'88*, Seite 67–72, 1988.

[FF93] Martin R. Frank und James D. Foley: *Model-Based User Interface Design by Example and by Interview.* In: *Proceedings of UIST '93, ACM Symposium on User Interface Software and Technology*, Seite 129–137, Nov. 1993.

[FF94] Martin R. Frank und James D. Foley: *A Pure Reasoning Engine for Programming by Demonstration.* Technical Report git-gvu-94-11, Georgia Institute of Technology, Graphics, Visualization and Usability Center, April 1994.

[FM93] Olaf Fricke und Daniel Moldt: *Formal Description of Structured Analysis Models by Coloured Petri Nets.* In: Gert Scheschonk und Wolfgang Reisig (Herausgeber): *Petri-Netze im Einsatz für Entwurf und Entwicklung von Informationssystemen, Informatik Aktuell*, Seite 165–179, 1993.

[Gam91] Erich Gamma: *Objektorientierte Software-Entwicklung am Beispiel von ET++: Design-Muster, Klassenbibliothek, Werkzeuge.* Dissertation, Universität Zürich, 1991.

[Gam99] Erich Gamma: *Design Patterns at Work.* Vortrag an der TU Darmstadt, verfügbar unter http://www.pu.informatik/docs/Gamma:Patternsatwork.ppt, Juli 1999. referenziert am 17.11.2000.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides: *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison Wesley, 1995.

[GHJV98] Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides: *Design Patterns CD: Elements of Reusable Object-Oriented Software.* Addison Wesley, März 1998.

[Gen00] Genlogic Inc.: *Glg Widget Sets.* http://www.genlogic.com/widgets.html, 2000. referenziert am 16.11.2000.

[Gli90a] Ephraim P. Glinert (Herausgeber): *Visual Programming Environments: Applications and Issues.* IEEE Computer Society Press, Los Alamitos, California, 1990.

[Gli90b] Ephraim P. Glinert (Herausgeber): *Visual Programming Environments: Paradigms and Systems*. IEEE Computer Society Press, Los Alamitos, California, 1990.

[GP96] T. R. Green und M. Petre: *Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework*. Journal of Visual Languages and Computing, (7) Seite 131–174, 1996.

[GBK+00] T. R. G. Green, M. M. Burnett, A. J. Ko, K. J. Rothermel, C. R. Cook und J. Schonfeld: *Using the Cognitive Walkthrough to Improve the Design of a Visual Programming Experiment*. In: *2000 IEEE International Symposium on Visual Languages*, Seite 172–179, September 2000.

[Gri00a] Richard Griffiths: *The Brighton Usability Pattern Collection*. http://www.it.bton.ac.uk/cil/usability/patterns/, 2000. referenziert am 15.9.2000.

[Gri00b] Richard Griffiths: *Poster des Workshops „Usability Patterns - Creating A Community“*. http://www.it.bton.ac.uk/staff/rng/UPLworkshop99/, 2000. referenziert am 26.10.2000.

[GPB00] Richard Griffiths, Lyn Pemberton und Jan Borchers: *Usability Pattern Language Workshop - The Poster*. http://www.it.bton.ac.uk/staff/rng/UPLworkshop99/Poster.html, 2000. referenziert am 29.11.2000.

[GPGW99] Tony Griffiths, Norman W. Parton, Carole A. Goble und Adrian J. West: *Task Modelling for Database Interface Development*. In: Hans-Jörg Bullinger und Jürgen Ziegler (Herausgeber): *Proceedings of HCI International '99, Munich, Germany*, Band 1, Seite 1033–1037. Lawrence Erlbaum Associates, August 1999.

[GMH98] J. C. Grundy, W. B. Mugridge und J. G. Hosking: *Visual Specification of Multi-View Visual Environments*. In: *Proceedings of the IEEE Symposium on Visual Languages*, Seite 236–243, 1998.

[Gud95] Birgit Guder: *Visualisierung von Smalltalk-Methoden*. Diplomarbeit, Technische Hochschule Darmstadt, Fachbereich Informatik, 1995.

[Haa87] Stefan Haas: *Visualisieren und Interpretieren von Petri-Netzen*. Technischer Bericht, Technische Hochschule Darmstadt, Fachbereich Informatik, FG Programmiersprachen und Übersetzer, 1987.

[HGCM87] Nick Hammond, Margaret M. Gardiner, Bruce Christie und Chris Marshall: *The role of cognitive psychology in user-interface design*. In: Margaret M. Gardiner und Bruce Christie (Herausgeber): *Applying Cognitive Psychology to User-Interface Design*, Seite 13–53. John Wiley and Sons, 1987.

[Han98] Hans Robert Hansen: *Wirtschaftsinformatik 1. Grundlagen betrieblicher Informationsverarbeitung*. Uni-Taschenbuch GmbH, Stuttart, 1998.

[Har01] Martin Hartmann: *Manuskript der Studienarbeit: Erweiterung von HotDraw-Editoren*. Technischer Bericht, Technische Universität Darmstadt, 2001.

[Hel94] Michael Helling: *DISPLAY - Vorschlag einer Vorgehensweise zur Bildschirmbeschreibung*. Diplomarbeit, Technische Hochschule Darmstadt, Fachbereich Informatik, Mai 1994.

[Hen97] Norman Hendrich: *Java für Fortgeschrittene*. Springer Verlag, 1997.

[HBvB+94] W. Hesse, G. Barkow, H. von Braun, H.-B. Kittlaus und G. Scheschonk: *Terminologie in der Softwaretechnik*. Informatik Spektrum, 17 Seite 96–105, April 1994.

[HP98] Hewlett-Packard Company: *HP VEE 5.0 Evaluation Kit - The visual programming language that helps you develop better tests faster*, 1998.

[HH92] William C. Hill und James D. Hollan: *Pointing and Visualization*. In: *CHI'92 Proceedings*, Seite 665–666. ACM, Mai 1992.

[Hir98] Robin Hirschl: *GPS-Signale und deren Verarbeitung*. http://www.ap.univie.ac.at/users/hirschl/gps/gps.html, 1998. referenziert am 9.9.2000.

[Hof97] Hans-Jürgen Hoffmann: *Vorlesungsunterlagen zu: Entwurf interaktionsfähiger Programme*. 1997.

[HV96] Andreas Homrighausen und Josef Voss: *Modellbasierte Entwicklung graphisch-interaktiver Anwendungen*. Softwaretechnik-Trends, Sonderheft mit den Beiträgen der GI-Fachtagung - Softwaretechnik 96, 16(3) Seite 121–128, September 1996.

[Hop97] Trevor Hopkins: *Objektorientierte Programmierung mit Smalltalk*. Carl Hanser Verlag München Wien, 1997.

[Hül97] Roland Hülscher: *Visual Constraint Rules*. Journal of Visual Languages and Computers, 8 Seite 425–451, 1997.

[IBM97a] IBM Corporation: *What is an object oriented user interface?* http://www.ibm.com/ibm/hci/guidelines/design/principles.html, 1997. referenziert am 12.11.2000.

[IBM97b] IBM Corporation: *What is an object oriented user interface?* http://www.ibm.com/ibm/hci/guidelines/design/principles.html, 1997. referenziert am 12.11.2000.

[IBM00] IBM Software: *Application Development: VisualAge for Java*. http://www-4.ibm.com/software/ad/vajava, 2000. referenziert am 28.1.2000.

[in-00a] in-integrierte informationssysteme GmbH: *Softwareentwicklungstool sphinx open - für dynamische Visualisierung von 2D-Grafik*. http://www.in-gmbh.de, 2000. referenziert am 20.3.2000.

[in-00b] in-integrierte informationssysteme GmbH: *TeleUse - grafische Bedieneroberflächen unter OSF/Motiv und Windows*. http://www.in-gmbh.de, März 2000. referenziert am 20.3.2000.

[Int00] Intelligent Instrumentation: *Visual Designer*. http://www.instrument.com/cat/software/designer/visual.htm, 2000. referenziert am 13.11.2000.

[Int96] Interactive Image: *Electronics Workbench - Das Elektroniklabor im Computer*, 1996. Werbebroschüre der Firma Interactve Image.

[IJK90] Tadao Ishikawa, Erland Jungert und Robert R. Korfhage (Herausgeber): *Visual Languages and Applications*. Plenum Press, New York, 1990.

[Jac00] Dirk Jacob: *Dokumentation des Anwendungsrahmens HotDraw mit Mustern*. Diplomarbeit, Technische Universität Darmstadt, Fachbereich Informatik, März 2000.

[JDM99] R. Jacob, L. Deligiannidis und L. Morison: *A Software Model and Specification Language for Non-WIMP User Interfaces*. Transactions on Computer-Human Interaction, 6(1) Seite 1–46, 1999.

[Jac92] Ivar Jacobson: *Object-Oriented Software Engineering*. ACM Press, 1992.

[JWZ93] Christian Janssen, Anette Weisbecker und Jürgen Ziegler: *Generierung graphischer Benutzungsschnittstellen aus Datenmodellen und Dialognetz-Spezifikationen*. In: Karl-Heinz Rödiger (Herausgeber): *Software Ergonomie '93*, Seite 335–344, 1993.

[Joh92] Ralph Johnson: *Documenting Frameworks using Patterns*. ACM Sigplan Notices, 27(10) Seite 63–76, 1992.

[Jos94] Nicolai Josuttis: *X Toolkit: Objektorientiertes Programmieren in C*. OBJEKTspektrum, (3) Seite 35–37, März 1994.

[Juh99] Ralph Juhnke: *Assistenten für Entwurfsmuster*. Diplomarbeit, Technische Universität Darmstadt, Fachbereich Informatik, Juli 1999.

[KPZ96] Jürgen Kahl, Jens Püttmann und Markus Zahnjel: *Evaluation eines Werkzeugs für Benutzungsschnittstellen*. Praktikum, Technische Hochschule Darmstadt, Fachbereich Informatik Fachgebiet Programmiersprachen und Übersetzer, 1996.

[Kah92] Ken Kahn: *Towards Visual Concurrent Constraint Programming*. Technischer Bericht SSL-91-92 (P92-00011), Xerox Parc, 1992.

[Kar95] Gabor Karsai: *A Configurable Visual Programming Environment*. IEEE Computer, Seite 36–44, März 1995.

[KLW92] S. Karsenty, J. A. Landay und C. Weikart: *Inferring graphical constraints with Rockit*. In: *HCI'92 Conference on People and Computers VII*, Seite 137–153. British Computer Society, September 1992.

[Käs99] Toni Käsbeck: *Teamwork am Tank.* IEE - Fachzeitschrift für Industrie-Automation, Industrielle Datentechnik, Steuerungs- und Regelungstechnik, Meßtechnik, Sensorik und Antriebstechnik, 44(S1) Seite 76–79, 1999.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Christina Videira Lopes, Jean Marc Loingtier und John Irwin: *Aspect-Oriented Programming.* In: *Proceedings of the ECOOP*, Seite 220–242. Springer-Verlag, Juni 1997.

[Kie95] Jan U. Kieß: *Objektorientierte Modellierung von Automatisierungssystemen.* Springer Verlag, 1995.

[KASC95] Takayuki Dan Kimura, Ajay Apte, Samudra Sengupta und Julie W. Chan: *Form/Formula, A Visual Programming Paradigm for User-Definable User Interfaces.* IEEE Computer, Seite 27–35, März 1995.

[Kit96] Barbara Kitchenham: *Evaluating Software Engineering Methods and Tools, Part 1: The Evaluation Context and Evaluation Methods.* Software Engineering Notes, 21(1) Seite 11–15, Januar 1996.

[Kno99] Knobloch GmbH: *Mini-Motor-Set Inhaltsliste.*
http://www.knobloch-gmbh.de/fischer/fi30342a.htm, 1999.
referenziert am 12.3.1999.

[Koc94] Marlis Koch: *Untersuchung von formalen Hilfsmitteln zur Modellierung der Dynamik von Objektmodellen.* Diplomarbeit, Technische Hochschule Darmstadt, Fachbereich Informatik, 1994.

[Kos94] Dirk Koschorek: *Visualization of Smalltalk Methods.* Interner Bericht, Fachbereich Informatik, FG Programmiersprachen und Übersetzer, Technische Hochschule Darmstadt, 1994.

[KP88] Glenn E. Krasner und Stephen T. Pope: *A Cookbook for using the Model-View-Controller User Interface Paradigm in Smalltalk-80.* Journal of Object Oriented Programming, August 1988.

[Kra97] Tom Krauß: *GUI-Frameworks in Smalltalk und Java.* OBJEKTspektrum, Seite 53–60, Mai 1997.

[Lak95] Charles Lakos: *From Coloured Petri Nets to Object Petri Nets.* In: Giorgio De Michelis (Herausgeber): *Theory and Applications of Petri Nets*, Springer Informatik, Seite 278–297. Springer Verlag, 1995.

[LM00] James A. Landay und Brad A. Myers: *Interactive Sketching for the Early Stages of User Interface Design.*
http://www.cs.cmu.edu/~ landay/home.html, 2000. referenziert am 14.11.2000.

[Lar99] Grant Larsen: *Using Patterns in the UML.* Communications of the ACM, 42(10) Seite 38–45, Oktober 1999.

[Lar92] James A. Larson: *Interactive Software: Tools for building user interfaces.* Prentice Hall, 1992.

[Lee00] Kenton Lee: *Technical X Window System and Motif WWW Sites.* http://www.rahul.net/kenton/xsites.html, 2000. referenziert am 11.10.2000.

[LE90] M. Leszak und H. Eggert: *Petri-Netz-Methoden und -Werkzeuge.* 1990.

[LMH97a] Xiaosong Li, Warwick B. Mugridge und John G. Hosking: *A Petri Net-based Environment for GUI Design.* In: *Proceedings SMC'97*, Band 3, Seite 2234–2239, Oktober 1997.

[LMH97b] Xiaosong Li, Warwick B. Mugridge und John G. Hosking: *A Petri Net-based Visual Language for Specifying GUIs.* In: *1997 IEEE Symposium on Visual Languages*, Seite 50–57, September 1997.

[LS93] Horst Lichter und Kurt Schneider: *vis-A-vis: Ein objekt-orientiertes Application Framework für grafische Entwurfswerkzeuge.* In: H.C. Mayr und R. Wagner (Herausgeber): *Objektorientierte Methoden für Informationssysteme*, Seite 187–207. GI, Springer Verlag, Juni 1993.

[Lie93a] Henry Lieberman: *Graphical Annotation as a Visual Language for Specifying Generalization Relations.* Proceedings of the IEEE Symposium on Visual Languages, Seite 19–24, 1993.

[Lie93b] Henry Lieberman: *Watch what I do*, Kapitel: *Making Programming Accessible to Visual Problem Solvers*, Seite 446–455. MIT Press Cambridge, London, 1993.

[LNHL00] James Lin, Mark W. Newman, Jason I. Hong und James A. Landay: *DENIM: Finding a Tighter Fit Between Tools and Practice for Web Site Design.* In: Thea Turner, Gerd Szwillus, Mary Czerwinski und Fabio Paterno (Herausgeber): *CHI 2000 Conference Proceedings*, Seite 510–517, 2000.

[LM00] Francisca Losavio und Alfredo Matteo: *Multiagent Models for Designing Object-Oriented Distributed Systems.* Journal of Object-Oriented Programming, Seite 8–12, Juni 2000.

[LCI+88] Frank Ludolph, Yu-Ying Chow, Dan Ingalls, Scott Wallace und Ken Doyle: *The FABRIK Programming Environment.* In: *Proceedings of the Workshop on Visual Languages, VL'88*, Seite 222–230, 1988.

[Mai98] Peter Maier: *PARTS for Java 2.5 - ein Paket für Profis.* OBJEKTspektrum, (4) Seite 70–76, April 1998.

[Man97] Theo Mandel: *Elements of user interface design.* John Wiley & Sons, Inc., 1997.

[Mar00] Ludger Martin: *Visualisierung von Komponenten für Benutzungsoberflächen.* Diplomarbeit, Technische Universität Darmstadt, Fachbereich Informatik, 2000.

[MS00] Ludger Martin und Elke Siemon: *Component Visualization Based on Programmer's Conceptual Models.* In: *Proceedings OOPSLA 2000 Companion*, Seite 73–74, 2000.

[McK91] P. J. McKerrow: *Introduction to Robotics.* Addison-Wesley Pub. Comp., 1991.

[MIC98] MICROGRAFIX: *Designer und die Phantasie nimmt Form an.* Werbebroschüre der Firma MICROGRAFIX, 1998.

[Mic00a] Microsoft: *It's Easy to Make Your Point-Anywhere.* Microsoft PowerPoint, November 2000.
http://www.microsoft.com/office/powerpoint, referenziert am 19.11.2000.

[Mic00b] Microsoft Corporation: *VisualBasic.*
http://www.visualbasic.about.com, 2000. referenziert am 18.11.2000.

[Mül85] Wolfgang Müller (Herausgeber): *Duden Bedeutungswörterbuch.* Duden Verlag, 1985.

[Mye88] Brad A. Myers: *Creating User Interfaces by Demonstration.* Perspectives in Computing. 1988.

[Mye90] Brad A. Myers: *Taxonomies of Visual Programming and Program Visualization.* Journal of Visual Languages and Computing, 1 Seite 97–123, 1990.

[Mye92] Brad A. Myers: *Languages for developing user interfaces*, Kapitel: *Introduction*, Seite 1–17. Jones and Bartlett, Boston, Mai 1992.

[Mye93] Brad A. Myers: *Watch what I do*, Kapitel: *Peridot: Creating User Interfaces by demonstration*, Seite 125–154. MIT Press, Pittsburgh, PA 15213, 1993.

[Mye95] Brad A. Myers: *User Interface Software Tools.* ACM Transactions on Computer-Human Interaction, 2(1) Seite 64–103, 1995.

[Mye96] Brad A. Myers: *User Interface Software Technology.* ACM Computing Surveys, 28(1) Seite 189–191, März 1996.

[Mye00a] Brad A. Myers: *Past, Present and Future of User Interface Software Tools.* ACM Transactions on Computer-Human Interaction, 7(1) Seite 3–28, März 2000.

[Mye00b] Brad A. Myers: *User Interface Software.* Course Description,
http://www.cs.cmu.edu/afs/cs.cmu.edu/user/bam/course/2000spring, 2000.
referenziert am 21.8.2000.

[Mye00c] Brad A. Myers: *User Interface Tools.*
http://www.cs.cmu.edu/~bam/toolnames.html, 2000. referenziert am 6.9.2000.

[MFM+96] Brad A. Myers, Alan Ferrency, Rich McDaniel, Robert C. Miller, Patrick Doane, Andy Mickish und Alex Klimovitski: *The Amulet V2.0 Reference Manual.* Technischer Bericht CMU-CS-166-R1, CMU-HCII-95-102-R1, Carnegie Mellon University, Februar 1996.

[MMK] Brad A. Myers, Richard G. McDaniel und David S. Kosbie: *Marquise: Creating Complete User Interfaces by Demonstration.*

[MMK93] Brad A. Myers, Richard G. McDaniel und David S. Kosbie: *Marquise: Creating Complete User Interfaces by Demonstration.* In: *INTERCHI'93: Human Factors in Computing Systems*, Seite 293–300, Februar 1993.

[NC87] M. Nanja und C. R. Cook: *An analysis of the online debugging process.* In: G.M. Olsen, S. Sheppard und E. Soloway (Herausgeber): *Empirical Studies of Programmers: Second Workshop*, Seite 172–184. Ablex, 1987.

[Nat00] National Instruments Corporation: *LabView.* http://www.ni.com, 2000. referenziert am 7.3.2000.

[NFS+93] Robert Neches, Jim Foley, Pedro Szekely, Piyawadee Sukaviriya, Ping Luo, Srdjan Kovacevic und Scott Hudson: *Knowledgeable development environments using shared design models.* In: *Proceedings of the international workshop on Intelligent user interfaces*, Seite 63–70, 1993.

[Oes97] Bernd Oestereich: *Objektorientierte Softwareentwicklung mit der Unified Modeling Language.* R. Oldenbourg Verlag, 1997.

[OS98] Bernd Oestereich und Susanne Strahinger: *UML-Design-Wettbewerb.* OBJEKTspektrum, September 1998.

[Ols98] Dan R. Olsen: *Developing User Interfaces.* Morgan Kaufmann Publishers, Inc., 1998.

[Ost98] Klaus Ostermann: *Programmieren mit Swing.* c't Heft 18, Seite 168–173, September 1998.

[Pal95] Massimo Paltrinieri: *A Visual Constraint Programming Environment.* In: Volker Haarslev (Herausgeber): *Proceedings of the 11th Symposium on Visual Languages, VL'95*, Seite 118–119, 1995.

[Par95] ParcPlace Systems: *VisualWorks Tutorial.* 1995.

[PAFJ00] Maria Pinto-Albuquerque, Manual J. Fonseca und Joaquim A. Jorge: *Visual Languages for Sketching Documents.* In: *2000 IEEE International Symposium on Visual Languages*, Seite 225–232, September 2000.

[Pöp96] Manfred Pöpping: *Objection - eine Entwicklungsumgebung für anwendungsspezifische Widgets.* Dissertation, Universität-Gesamthochschule Paderborn, Mai 1996.

[Pos96] Jörg Poswig: *Visuelle Programmierung.* Hanser Verlag, 1996.

[PRO92] PROMATIS: *Modellierung von Systemverhalten mit Petri-Netzen.* Broschüre der Firma PROMATIS Informatik GmbH & Co. KG, Straubenhardt, 1992.

[Pue99] Angel R. Puerta: *Human-Centered Model-Based Interface Development.* In: Hans-Jörg Bullinger und Jürgen Ziegler (Herausgeber): *Proceedings of HCI International '99, Munich, Germany*, Band 1, Seite 1048–1052. Lawrence Erlbaum Associates, August 1999.

[PS94] Angel R. Puerta und Pedro Szekely: *Model-Based Interface Development.* Tutorial CHI '94, 1994.

[RDO98] D. Janaki Ram, R. A. Dwivedi und Ramakrishna Ongole: *An Implementation Mechanism for Design Patterns.* Software Engineering Notes, 23(5) Seite 52–56, September 1998.

[Rat00] Rational Software Corporation: *Rational Rose.* http://www.rational.com/products/rose, 2000. referenziert am 15.8.2000.

[Rei85] Wolfgang Reisig: *Systementwurf mit Netzen.* Springer-Verlag, Berlin; Heidelberg; New York; Tokyo, 1985.

[RS00] Michael Renker und Torsten Scheidler: *Manuskript der Programmdokumentation für das Projekt Entwurf eines Editors zur Realisierung des MVC-Konzeptes.* Technischer Bericht, Technische Universität Darmstadt, 2000.

[RC93] Alex Repenning und Wayne Citrin: *Agentsheets: Applying Grid-Based Spatial Reasoning to Human-Computer Interaction.* In: *Proceedings of the IEEE Symposium on Visual Languages*, Seite 77–82, August 1993.

[Rös00] Kerstin Röse: *Der Mensch - ein Risikofaktor.* In: Detlef Zühlke (Herausgeber): *Teilnehmerunterlagen zum Seminar Mensch-Maschine-Interaktion in komplexen technischen Systemen*, 2000.

[RBP+91] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy und William Lorensen: *Object-Oriented Modeling and Design.* Prentice-Hall, 1991.

[RW92] Andrew Rutherford und John R. Wilson: *Models in the mind*, Kapitel: *Searching for mental models*, Seite 195–223. Academic Press, 1992.

[Sch97a] Stefan Schiffer: *Visuelle Programmierung.* Addison Wesley, 1997.

[Sch97b] Hans-Jochen Schneider: *Lexikon der Informatik und Datenverarbeitung.* R. Oldenbourg Verlag, 4. Auflage, 1997.

[Sch95] Ute Schneider: *GUI-Builder im praktischen Einsatz - Mehr oder weniger Arbeit.* iX, 3 Seite 50–55, 1995.

[Sch92] Bruno Schrade: *Petrinetze als Programmiersprache.* Softwaretechnik-Trends, (1) Seite 42–52, Februar 1992.

[Sch96] Martin Schwarz: *Vergleich von Verbunddokumentarchitekturen und Verbunddokumentframeworks, Implementierung einer verteilten Mehrbenutzereditorkomponente.* Diplomarbeit, Technische Universität München, 1996.

[Sci00] Scientific Computers GmbH: *MATLAB.* http://www.scientific.de, September 2000. referenziert am 13.9.2000.

[Sel99] Bran Selic: *Turning Clockwise. Using UML in the Real-Time-Domain.* Communications of the ACM, 42(10) Seite 46–54, Oktober 1999.

[SGW94] Bran Selic, Garth Gullekson und Paul T. Ward: *Real-Time Object-Oriented Modelling.* Wiley Professional Computing, 1994.

[SM92] Sally Shlaer und Stephen J. Mellor: *Object Lifecycles: Modeling the World in States.* Prentice-Hall, 1992.

[Shn83] Ben Shneiderman: *Direct Manipulation: A Step Beyond Programming Languages.* IEEE Computer, 16(8) Seite 57–69, September 1983.

[Shn98] Ben Shneiderman: *Designing the User Interface.* Addison-Wesley Longman, Inc., 1998.

[Shu88] Nan C. Shu: *Visual Programming.* Van Nostrand Reinhold Company Inc., 1988.

[Sie92] Elke Siemon: *Methoden zur Objektorientierten Analyse: Vergleich und Fallbeispiel.* Diplomarbeit, Technische Hochschule Darmstadt, Fachbereich Informatik, November 1992.

[Sie96] Elke Siemon: *Graphical Specification for User Interfaces in Automation Systems.* Vorgestellt beim Workshop „Basic Research“ der CHI'96, 1996. verfügbar unter http://www.pu.informatik.tu-darmstadt.de/siemon/Veroeffentlichungen.

[Sie00] Elke Siemon: *Pattern language proposal and position statement.* CHI'2000 Workshop on Patterns for Interaction Design, Januar 2000.

[SBG00] Elke Siemon, Alan Blackwell und Thomas Green. Der Begriff *trapdoor* entstand in einer persönlichen Diskussion am 8.9.2000, 2000.

[SBF96] Elke Siemon, Jürgen Buchner und Martin Frisch: *Teamentwurf von Benutzungsschnittstellen unter Verwenden von Agenten, basierend auf einem verteilten MVC.* In: *STAK '96 Softwaretechnik in Automation und Kommunikation - Rechnergestützte Teamarbeit*, ITG Fachbericht 137, Seite 45–56, März 1996.

[SJ99] Elke Siemon und Ralph Juhnke: *Tools for implementing design patterns.* In: *OOPSLA'99 Companion*, Band 34 von *ACM SIGPLAN Notices*, Seite 81–82, Oktober 1999.

[SS96] Elke Siemon und Claudia Seeger: *An Empirical Study on Differences in User Interface Specification Behavior between Computer Scientists and Electrical Engineers.* vorgestellt als Late Breaking Research Poster, CHI'96, 1996. verfügbar unter http://www.pu.informatik.tu-darmstadt.de/siemon/Veroeffentlichungen.

[SW00] Elke Siemon und Yongmei Wu: *On Techniques for Visual Task-oriented End User Programming.* Beitrag zum Workshop „The Visual End User“, IEEE Symposium for Visual Languages, September 2000.

[Slo98] Arthur Sloman: *Diagrams in the Mind?* Invited Talk, Thinking with Diagrams Conference, Abersytwyth,
http://www.aber.ac.uk/~plo/TWD98, 1998. referenziert am 30.8.1998.

[Sou97] Jiri Soukup: *Implementing Patterns.* In: James O. Coplien und Douglas C. Schmidt (Herausgeber): *Pattern Languages of Program Design, PLoP'97*, Band 1, Seite 395–415, 1997.

[SC95] Scott B. Steinman und Kevin G. Carver: *Visual Programming with Prograph CPX.* Manning Publications Co., 1995.

[Ste97] Uwe Steinmüller: *Let it swing.* iX, (10) Seite 132, 1997.

[Str97] Alois Stritzinger: *Komponentenbasierte Softwareentwicklung.* Addison-Wesley Longman, 1997.

[Suk93] Piyawadee „Noi“ Sukaviriya: *A Second Generation User Interface Design Environment: The Model and The Runtime Architecture.* In: *INTERCHI '93 Conference Proceedings*, 1993.

[SLN93] Pedro Szekely, Ping Luo und Robert Neches: *Beyond Interface Builders: Model based Interface Tools.* In: *Proceedings of INTERCHI '93*, Seite 383–390, April 1993.

[Szw92] Gerd Szwillus: *Graphische Constraints - Syntaxregeln für graphische Darstellung.* Vortrag in Dortmund am 26.11.1992, 1992.

[SB99] Gerd Szwillus und Birgit Bomsdorf: *Tool Support for Task-Based User Interface Design.* CHI '99 Workshop Description, http://www.uni-paderborn.de/fachbereich/AG/szwillus/chi99/ws/, 1999. referenziert am 23.10.1999.

[Szy99] Clemens Szyperski: *Component Software - Beyond Object-Oriented Programming.* Addison Wesley, 1999.

[Thi94] Markus A. Thies: *Adaptive User Interfaces.* In: K. Brunnstein und E. Raubold (Herausgeber): *13th World Computer Congress 94*, Band 2, Seite 196–201. IFIP, Elsevier Science B.V., 1994.

[Tid00] Jenifer Tidwell: *Common Ground: A Pattern Language for Human-Computer Interface Design.* http://www.mit.edu/~jtidwell/interaction_patterns.html, 2000. referenziert am 12.1.2000.

[TB87] Heinz Töpfer und Peter Besch: *Grundlagen der Automatisierungstechnik.* Carl Hanser Verlag, 1987.

[Uni98] Universität der Bundeswehr: *Computerlexikon.* http://christine.unibw-hamburg/Book, 1998. referenziert am 13.7.1998.

[Vää95] Kaissa Väänänen: *Metaphor-Based User Interfaces for Information Authoring, Visualization and Navigation in Multimedia Environments.* Dissertation, Technische Hochschule Darmstadt, 1995.

[Vie89] Axel Viereck: *Eine Modellierung von Mensch-Rechner-Dialogen durch strukturierte, markierte Netze.* Notizen Interaktives Programmieren, Seite 73–81, 1989.

[WB99] Renate Wahrig-Burfeind: *Fremdwörterlexikon.* Bertelsmann Lexikon Verlag, 1999.

[WC99] Kathy Walrath und Mary Campione: *The JFC Swing Tutorial: A Guide to Constructing GUIs.* The Java Series. Addison-Wesley, 1999.

[Wan93] Jens Wandmacher: *Software Ergonomie.* de Gruyter, 1993.

[Wel00] Martijn Welie: *The Amsterdam Collection of Patterns in User Interface Design.* http://www.cs.vu.nl/~martijn/patterns, 2000. referenziert am 5.9.2000.

[Wis93] Wissenschaftlicher Rat der Dudenredaktion (Herausgeber): *Duden Informatik*. Dudenverlag, 1993.

[Wol00] Wolfram Research Inc.: *Building Systems with Mathematica*. http://www.wolfram.com/products/mathematica/tour, 2000. referenziert am 17.9.2000.

[Wu98] Yongmei Wu: *Design models and object-oriented framework for user interfaces in computer-supported control technology*. Arbeitsbericht, März 1998.

[ZZ92] Alfred Zeidler und Rudolf Zellner: *Software Ergonomie*. R. Oldenbourg Verlag, 1992.

[Zha98] Da-Qian Zhang: *VisPro: A Visual Language Generation Toolset*. IEEE Software, Seite 195–202, 1998.

[Zha97] Phillip Zhang: *Ein Vektorgrafikeditor für HotDoc*. Studienarbeit, Technische Hochschule Darmstadt, Fachbereich Informatik, FG Programmiersprachen und Übersetzer, 1997.

[Züh00] Detlef Zühlke: *Mensch-Maschine-Systeme der Zukunft*. Seminarunterlagen des Seminars Mensch-Maschine-Systeme der Zukunft, Kaiserslautern, Mai 2000.